Universitat Politècnica de Catalunya

Master in Innovation and Research in Informatics

Data reuse design exploration in OmpSs@FPGA

Author:
Marc Mateu Sebastián

Supervisor:
Daniel Jiménez González
Co-supervisor:
Xavier Martorell Bofill

October 17, 2019

Abstract

Due to their characteristics, Field Programmable Gate Arrays (FPGAs) are nowadays widely used to accelerate specific parts of applications. In this thesis, the OmpSs@FPGA tool chain has been extended to try to reduce the overall communication time due to copies of data when it is possible to reuse data already in the BRAM of the accelerators.

Acknowledgments

I would like to thank all that people that directly gave me support and patiently helped me to do this project, Dani, Antonio, Miquel and Jaume, and to my family and friends that indirectly made this work easier for me. Without all of you this project would not have been possible.

This work is supported by the Spanish Government (projects SEV-2015-0493 and TIN2015-65316-P), by the Generalitat de Catalunya (2017-SGR-1414 and 2017-SGR-1328) and by the European Research Council (RoMoL GA 321253). We also thank the Xilinx University Program.

Contents

Chapter 1

Introduction

Due to their characteristics, Field Programmable Gate Arrays (FPGAs) are nowadays widely used to accelerate specific parts of applications. They can be reprogrammed to implement any possible logic, which helps the process of verification and evaluation, and also designers can take benefit from their parallel architecture to achieve improvements in the performance. Because of this, FPGAs are becoming an alternative to Graphics Processing Units (GPUs)[1][2][3], even with a better energy throughput[4].

Normally FPGAs consist of four main components: programmable logic blocks which implement logic functions; programmable routing that connects these logic functions; I/O blocks that are connected to logic blocks through routing interconnect and make off-chip connections; and Block Random Access Memory (BRAM)[5]. The diagram of the basic FPGA structure and internal components is shown in Figure 1.

As a result, companies and research centers are focusing more on systems and applications which involve FPGAs. For example, graphic processing has been accelerated to improve the experience in autonomous car driving[6] or medical interventions with 3D images[7], but also other fields such as cryptography[8], security[9] and even data centers have taken benefit from FPGAs[10]. Apart from that, research has also been done to enhance their reliability and ease the integration with other systems[11].

Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS)[12] is one of these involved research centers, and this project has been developed as an extension of its particular programming model, OmpSs[13]. Besides, it is also part of the EuroEXA European project which aims to provide the template for an upcoming exascale system with FPGA acceleration for computational, networking and storage operations[14].

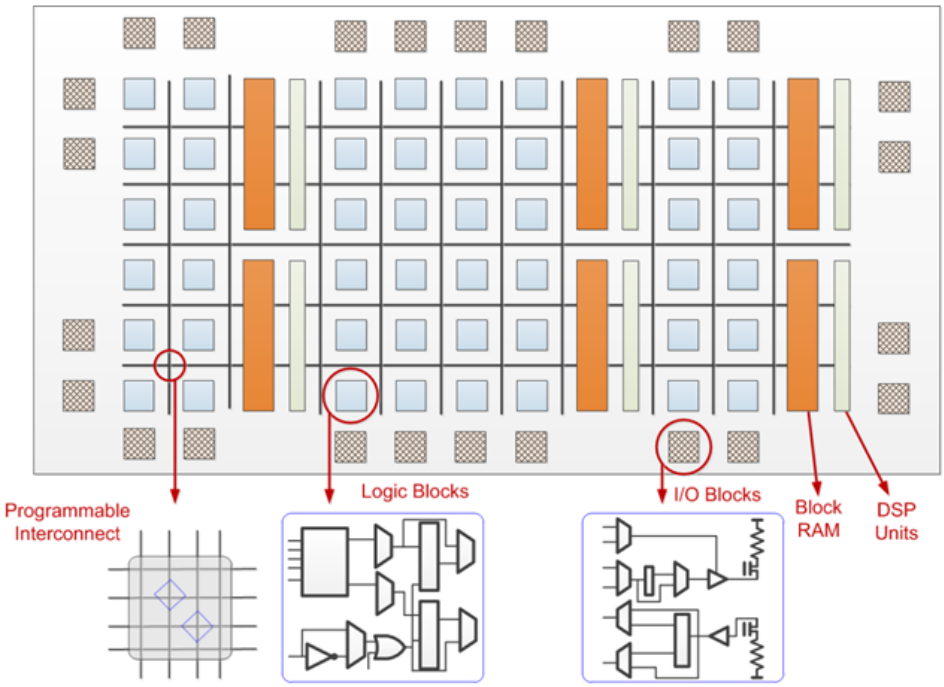

Figure 1: Diagram of the basic FPGA structure and internal components

Source: Medium: What are FPGAs?

The OmpSs objective is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity. This is achieved by the use of data-dependencies between the different tasks of the program and the target construct, which allows to execute tasks in specific architectures such as GPUs, SMP and FPGAs. However, since the process of targeting FPGA is more complex than other devices due to aspects like synthesis, block design or communication, a separate group, OmpSs@FPGA, focuses to support easily offloading of tasks to FPGA devices in an automatic and transparent manner to the programmer[15].

During the FPGA task offloading configuration there are several parts that can be optimized to improve the performance. They can be classified more or less in software-related, such as the communication between the different components of the heterogeneous system and the scheduling of tasks, and hardware-related, such as applying techniques using High Level Synthesis (HLS) to exploit the inherent parallelism of the FPGA. Besides, depending on the FPGA and the board targeted some details like frequency

and address range may differ.

The OmpSs@FPGA ecosystem provides an automatic infrastructure to execute the original program in the host device and accelerate the requested task in the FPGA. The programmer has to indicate the direction of the task parameters (IN, OUT or INOUT), and then it is converted to, at compile time, an IP (also referred to it as kernel accelerator) and placed in the block diagram of the Programmable Logic part of the FPGA (PL). During the execution of the program, the runtime of the OmpSs programming model will be responsible of submitting the tasks, as well as synchronizing with the finalization.

Figure 2 shows an execution trace of a matrix multiplication using OmpSs@FPGA. It is configured with two SMP threads (the first two rows) that are responsible of submitting the FPGA tasks to the kernel accelerators, which is depicted in red color. On the other hand, two kernels accelerators are being used (the last two rows) to execute the FPGA tasks. Their execution process is divided in three steps: copying in data from the host (in green), executing the FPGA task (in brown) and copying out data to the host (in blue).

Figure 2: Execution trace of a matrix multiplication using OmpSs@FPGA

As described in [16], the basic flow is to fetch data from external memory to on-chip buffer (BRAM), and then feed them into registers and Processing Engines (PEs). After the PE computation completes, results are transferred back to on-chip buffers and to the external memory if necessary. Even though BRAMs are expensive and have small storage capacity, they can handle random accesses with much higher performance than the off-chip

DRAM, which favors only sequential or predictable access patterns[17]. Besides, using local BRAMs in the kernel accelerator allows the programmers to apply HLS techniques such as data structure partitioning to improve the overall application performance by accessing, in parallel, to the data structures.

In Figure 3 a general overview of the different memory spaces that an input and/or output data of a FPGA task may be placed is shown. It has to be noticed that the data must be placed in the kernel space of the DRAM memory to be accessible by the FPGA. As a general rule, once the input data is transferred to the internal BRAM, the computation of the task can start; when it finishes, the output data is transferred back to the kernel space memory.

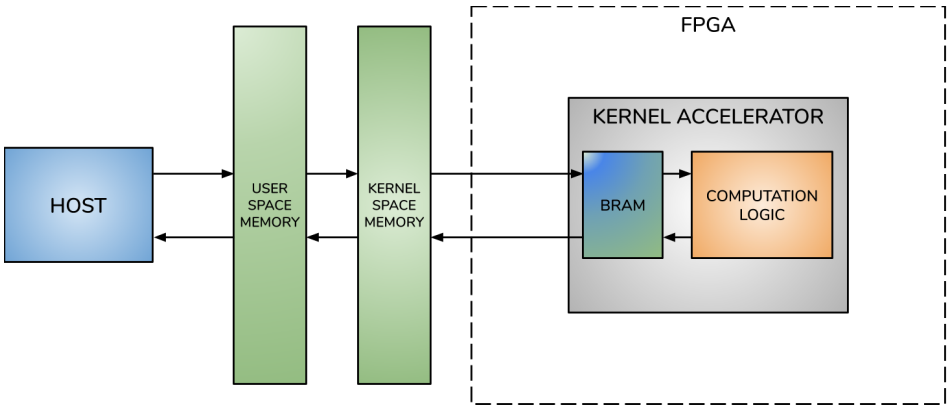

Figure 3: General overview of the different memory spaces that an input and/or output data of an FPGA task may be placed.

Even though the OmpSs@FPGA group is constantly working to enhance the automatic task offloading to FPGA accelerators, there are still several challenges. One of them is the memory transfers between the host and the FPGA accelerator; at this moment, the data transfers between user space and kernel space memory are optimized to be done only when they are needed. However, the copies between the kernel space and the internal BRAM of the kernel accelerator are always performed, regardless if it is necessary to do it because, for example, the input data is already on it. As can be seen in Figure 2, depending on the application being accelerated the memory transfers can be the bottleneck, whereas the computation can be executed rapidly if the FPGA parallelism is correctly exploited. Therefore, this project focuses on trying to reduce the overall communication time due

to copies of data when it is possible to reuse data already in the BRAM of the accelerators. In order to achieve it, several mechanisms have been developed to analyze and compare their impact in the performance as well as the resource utilization.

The remainder of this document is organized as follows: Section 2 presents the state of the art. Section 3 describes the experimental setup chosen, such as the hardware/software resources and the benchmarks evaluated. Section 4 shows a preliminary analysis performed in order to decide the strategy to mitigate the problem presented on this project. In Sections 5 and 6 the different explored designs and their details are presented. The experimental results of these designs when executing the benchmarks are evaluated in Section 7, which is followed by conclusions and future work in Section 8.

Chapter 2

State of the art

This section presents the related work that has been done concerning the topic of this project. To this end, the state of the art has been explored from two point of views; firstly, from the automatic task offloading to FPGA accelerators, and secondly, from the methods employed to improve FPGA executions performance.

On the one hand automatic task offloading to FPGA accelerators has been explored, as well as applying techniques to enhance the performance. However, even though memory transfers have been taken into account in the majority of the research, the data reuse proposed in this project does not appear to have been explored. Sommer, Korinth, and Koch [18], similar to OmpSs@FPGA, use OpenMP target directives to offload tasks to FPGA-based accelerators on the existing Clang infrastructure. Rigamonti et al. [19] propose an automated framework that allows the transparent execution of ordinary code on a heterogeneous platform including an FPGA, and dynamically adapts its behavior to the execution scenario and workload of the system. Besides, constant inputs are retained throughout the computations to reduce the amount of data to transfer. Patyk et al. [20] propose a design method, which uses the Transport Triggered Architecture (TTA) processor template and the TTA-based Co-design Environment toolset to automate the design process. Álvarez et al. [21] introduce a new offloading methodology which allows both large compatibility with different device architectures and flexibility in the design of the computation kernels using OpenMP but not its compiler. In order to support it, a flexible and interoperable developed runtime infrastructure fully integrates with the standard OpenMP runtime. Finally, Knaust, Mayer, and Steinke [22] propose to use OpenMP target offloading making use of the preexisting OpenCL SDK of the FPGA

vendor. However, it concludes that unnecessary transfers were done, and both read and write FPGA memory transfers could be reduced by half in future work.

On the other hand, considerable research has been done with FPGA architecture to improve the performance of time consuming applications, and some of them consider data reuse. Zhang and Li [23] propose a kernel design to implement Convolutional Neural Network accelerators using OpenCL FPGA that takes into account the memory bandwidth. While it improves the computational resources utilization by increasing the on-chip data reuse, it also minimizes the random data access penalty from external memory. Yuxin Wang et al. [24] present an automated optimization flow (AMO) that combines memory partitioning and merging with data reuse and pipelining for FPGA behavioral synthesis. This AMO is applied to the accelerated function to improve its memory accesses, and the data reuse is achieved by using a reuse buffer. Besides, Becker et al. [25] propose an interesting function reuse-based technique for soft-core processors where, each time a function executes, its results are dynamically stored in a BRAM Reuse Table (RT) and, when the same function with the same input arguments is called again, the output can be directly fetched, avoiding re-calculation and improving performance.

Chapter 3

Experimental setup

In this chapter the experimental setup decided to develop the project is presented. Basically, there are four different aspects: hardware, software tools, benchmarks used to perform the tests and the methodology applied to these tests.

3.1 Hardware

The hardware environment for developing and evaluating the different implementations has been the ZC706 evaluation board with a Zynq®-7000 XC7Z045-2FFG900C SoC [26]. The relevant board features for this project are as follows:

- Zynq-7000 XC7Z045-2FFG900C SoC
  - 1090 18Kb BRAMs
  - 900 DSP48E
  - 437200 Flip-Flops
  - 218600 Look-up tables
- Two ARM® Cortex™-A9 MPCore™ Application Processor Units at a maximum frequency of 800 MHz
- 1 GB DDR3 memory SODIMM on the programmable logic (PL) side
- 1 GB DDR3 component memory (four [256 Mb x 8] devices) on the processing system (PS) side

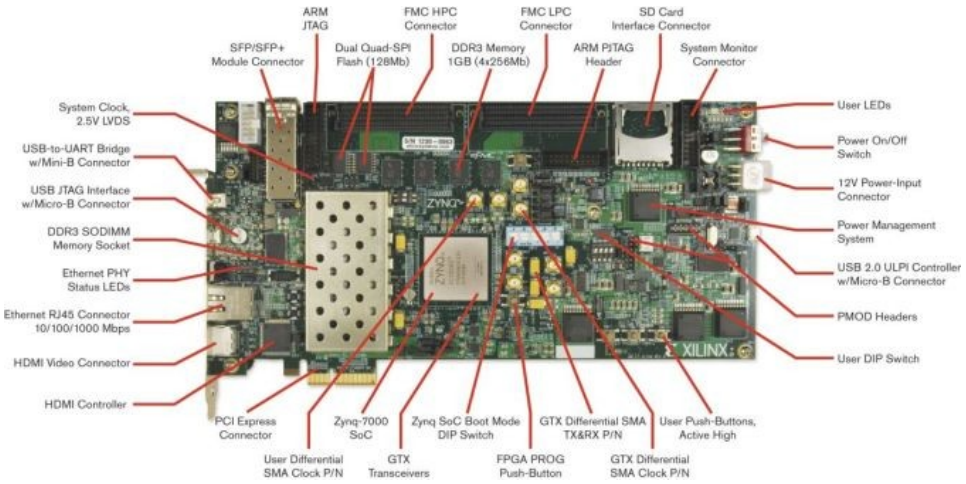

Figure 4: ZC706 evaluation board and its components

In Figure 4 and 5 the ZC706 evaluation board components and its block diagram can be seen, respectively.

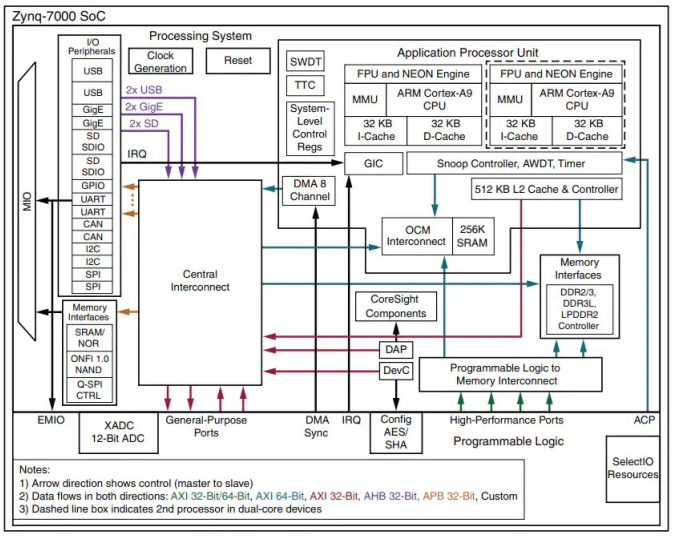

Figure 5: ZC706 block diagram

3.2 Software tools

As indicated in the previous chapter, the OmpSs@FPGA toolchain has been used as the base system to offload tasks to the FPGA. It is composed by three main components: the OmpSs compiler (Mercurium [27]), a source-to-source compiler which provides the necessary support for transforming the high-level directives into a parallelized version of the application; the OmpSs runtime (Nanos++ [28]) which provides the parallel services to manage all the parallelism in the user-application, including task creation, synchronization and data movement, and provide support for resource heterogeneity; and autoVivado, a tool that uses the Xilinx Vivado Design Suite software [29] to automatically manage the bitstream generation process, including steps such as synthesis of the HLS code, integration of the different IPs and the Processing System with a interconnection network and device-tree generation. The structure of the components involved in the application binary and FPGA bitstream generation are shown in Figure 6.

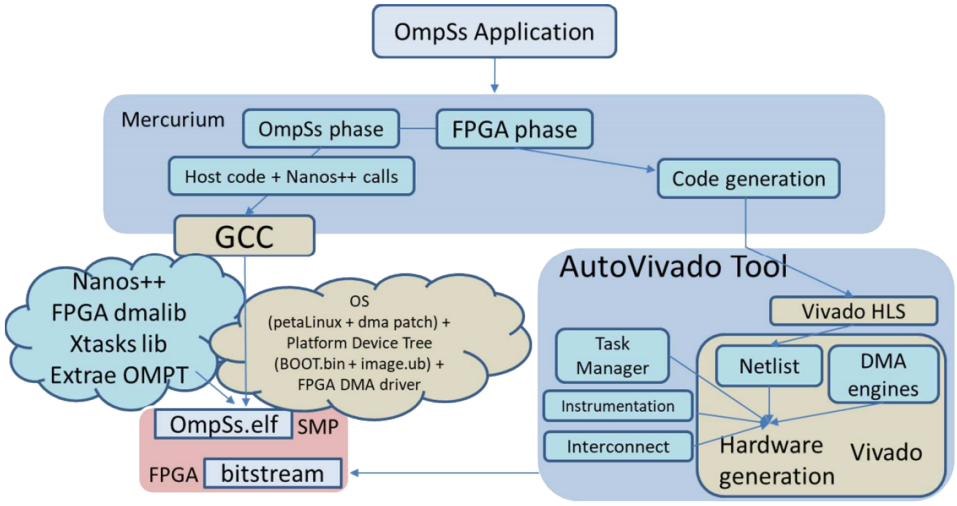

Figure 6: Structure of the components involved in the application binary and FPGA bitstream generation

The tasks with fpga target device are isolated by Mercurium into separate files, which are used by autoVivado tool to generate the FPGA bitstream. Moreover, autoVivado uses different TCL scripts to generate a project for the Xilinx tools that allows Nanos++ interact with the FPGA accelerators. Also, there is support for instrumenting the FPGA accelerators and integrate the information into the Extrae tool [30] which, complemented

with the Paraver tool [31], are used for trace visualization.

Regarding the software versions, OmpSs@FPGA release 1.3.2, which includes all the necessary tools, and Vivado 2017.3 have been used since they are stable and compatible between them.

3.3 Benchmarks

In order to verify and evaluate the performance of the implementations, two real applications has been tested, which are the common matrix multiplication and a N-Body.

Concretely, the matrix multiplication application has been executed in blocks of 64∗64 elements size, and each block calculation has been annotated as an FPGA task, which has A and B matrix blocks as inputs, and C matrix block as input and output. Apart from that, the three matrix multiplication loops have been tuned to test different cases, which are:

- C loop as the inner-most loop. This is the most frequent implementation of the matrix multiplication and aims to execute consecutive tasks with the same C block, but during the execution it is possible the interpolation of other tasks with different C blocks.
- C loop as the inner-most loop with a taskwait just after it. It aims to force the execution of consecutive tasks with the same C block.
- A loop as the inner loop. It aims to execute consecutive tasks with the same A block, but during the execution it is possible the interpolation of other tasks with different A blocks.
- A loop as the inner-most loop with a taskwait just after it. It aims to force the execution of consecutive tasks with the same A block.

With this, the variation in performance depending on the input and/or output data reuse can be clearer to visualize.

3.4 Methodology

To compare the developed implementations with the current in OmpSs@FPGA, the execution time from the first to the last FPGA offloaded task of each benchmark has been considered. This time has been obtained as the mean of five executions, and additionally the result has been checked against one

without FPGA tasks to verify the correctness. Besides, the FPGA has been designed with a frequency of 100MHz.

Finally, to visualize graphically if the implementations are exploiting the data reuse, two methods have been used. On the one hand, the Paraver tool which gives a global perception of the application behavior, including the host part (the submission of the FPGA tasks and the rest of the code) and the FPGA part (copies and computation). On the other hand, the Integrated Logic Analyzer (ILA) [32] from Vivado, which monitors the internal signals of the FPGA during the execution and show them as a waveform.

Chapter 4

Preliminary analysis

Before starting to design mechanisms to try to reuse FPGA data, a preliminary analysis on the components involved with memory transfers has been done. This analysis considers the OmpSs@FPGA toolchain and hardware details of the board used, and will be useful to take decisions later.

4.1 OmpSs programming model

First of all, it is important to explain the work and data flow between tasks in the OmpSs programming model since it will be taken into account during the mechanisms implementation.

As indicated previously, OmpSs taskifies regions of code annotated by the user. The data-dependencies among tasks are expressed using the in, out and inout clauses, which allow to specify what data a task is waiting for and signaling its readiness. Each time a new task is created, its in and out dependencies are matched against those of existing tasks and, if a dependency, either RaW, WaW or WaR, is found the task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime, and tasks are scheduled for execution as soon as all their predecessor in the graph have finished (which does not mean they are executed immediately) or at creation if they have no predecessors.

4.2 OmpSs@FPGA

Figure 7 sums up graphically all the different OmpSs@FPGA components involved in memory transfers.

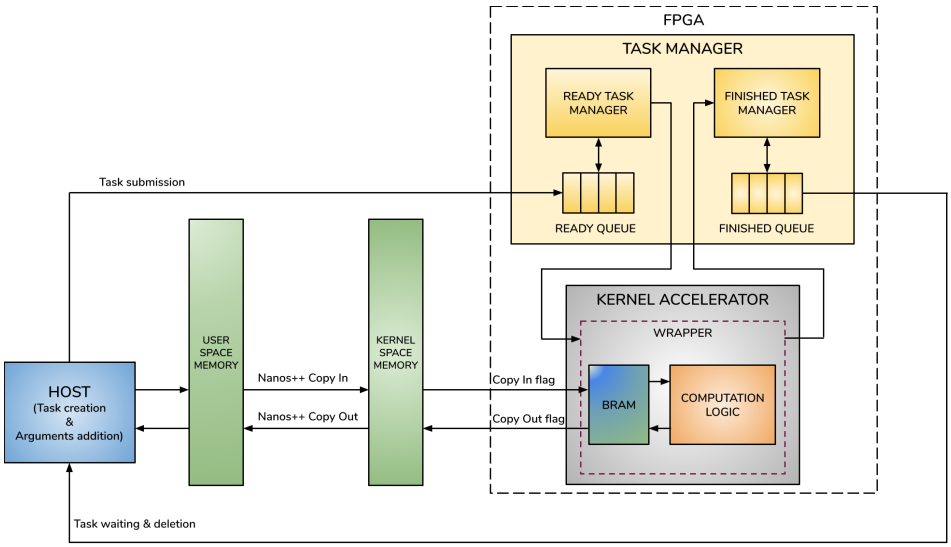

Figure 7: OmpSs@FPGA relevant components concerning memory transfers.

4.2.1 Nanos++ and xTasks

The OmpSs@FPGA software part is responsible of managing the FPGA tasks, which includes steps such as creation, addition of the arguments, submission or deletion. This process is performed by Nanos++, but it also relies on xTasks, an external library containing all the functions related to FPGA tasks to simplify the implementation of the FPGA plugin.

Regarding the memory transfers, Nanos++ moves data between user and kernel space memory depending on the inputs and outputs of the FPGA task. Furthermore, these transfers create a mapping between both space memories and are already optimized to be performed only in 4 cases:

- When new data has to be transferred to kernel space memory.
- When mapped data is modified in the user space by another device (for example, SMP) and a new FPGA tasks has it as input.
- When there is a taskwait, all the mapped data must be transferred from kernel to user space memory.
- When another device has as input the output data of an FPGA task which is located in kernel space memory.

On the other hand, xTasks indicates if data has to be moved between the kernel accelerator and the kernel space memory, depending on the inputs and outputs of the FPGA task. This is achieved at the moment of adding its arguments, which starts with a task header that involves information such as the task ID, and continues with the arguments of the data to be transferred to the accelerator, divided as follows:

- Arguments addresses, which indicates the kernel space memory addresses that the kernel accelerator must access to read/write the corresponding arguments.
- Arguments flags, which indicates if an argument data must be read or written from/to kernel space memory.

Therefore, the arguments flags play a key role in the purpose of this project since they can be deactivated to avoid unnecessary copies. In the OmpSs@FPGA version used there is no logic concerning this detail and these flags are always activated depending on the data direction of the argument. Besides, once the arguments are sent to the kernel accelerator they are managed by a special function called wrapper.

4.2.2 Accelerator wrapper

As explained previously, in the automatic process of the hardware configuration the task that is accelerated in the FPGA is encapsulated as an IP. However, the task function is also extended with a wrapper function that is responsible of managing aspects such as data copies or instrumentation features. A general view of the wrapper function is shown in Algorithm 1:

Algorithm 1 Accelerator wrapper

```
1: Read the task header
2: for all Task arguments do
3:    Read the argument flags
4:    Read the argument address
5:    if IN copy flag activated then
6:       Copy the data from kernel space to BRAM memory
7:    end if
8: end for
9: Call the task function
10: for all Task arguments do
11:    if OUT copy flag activated then
12:       Copy the data from BRAM memory to kernel space
13:    end if
14: end for
15: Indicate the task has finished
```

Additionally, when the wrapper finishes (meaning the task has been executed) it has to notify it to let the accelerator execute new tasks and schedule possible successor tasks. Therefore, there is another IP that schedules the ready tasks to kernel accelerators when they are free, called TaskManager.

4.2.3 TaskManager

The TaskManager is an IP that manages the flow of the FPGA tasks sent by the Nanos++ runtime from the host. It involves several components, but the main actors are the ReadyTaskManager and the FinishedTaskManager. Both of them act like intermediate processes between the host and the kernel accelerators, and to this end each of them share a specialized BRAM with the host.

Inside the FPGA, the acceleration flow can be divided in three steps. Firstly, the ReadyTaskManager checks if Nanos++ has added FPGA tasks that are ready to be executed (because they do not have any dependency with predecessor tasks) in a BRAM called ReadyQueue. This BRAM is basically a matrix that contains 32 ready task entries for each accelerator, and the entries are liberated once their tasks are sent to the kernel accelerator. A pseudo-code of the ReadyTaskManager is shown in Algorithm 2.

Secondly, the task sent to an accelerator kernel is executed. Lastly, the third step starts when the wrapper indicates its finalization, as mentioned before. The FinishedTaskManager receives the data and marks the task as

Algorithm 2 ReadyTaskManager

```
1: for all Accelerators do
2:    acceleratorEntry ← 0
3: end for
4: acceleratorIndex ← 0
5: while true do
6:    if Task in ReadyQueue[acceleratorEntry[acceleratorIndex] is ready
      and accelerator is available then
7:       Read task
8:       Send task to the accelerator
9:       Mark the entry as free
10:      Mark the accelerator as busy
11:      acceleratorEntry[acceleratorIndex] + +
12:   else
13:      acceleratorIndex + +
14:   end if
15: end while
```

finished in its own BRAM, called FinishedQueue, that works in a similar manner as the ReadyQueue. Besides, it also updates the state of the kernel accelerator as available. A pseudo-code of the FinishedTaskManager is shown in Algorithm 3.

Algorithm 3 FinishedTaskManager

```
1: for all Accelerators do
2:    acceleratorEntry ← 0
3: end for
4: while true do
5:    Read task information from the wrapper
6:    Mark the task as finished in finishedQueue[acceleratorEntry[accID]
7:    Mark the accelerator as available
8:    acceleratorEntry[accID] + +
9: end while
```

4.3 Board memory transfers

Additionally, an analysis of the different available memories in the board used has been performed to explore if the data flow of the input and output

data of tasks, inside the FPGA, can be optimized.. For instance, when a kernel accelerator has to copy out some data, instead of copying it to the kernel user space memory it could be possible to place it in a BRAM shared by all the kernel accelerators, and if later a different kernel accelerator has that data as input it could read it from there. Therefore, this BRAM would be used as a cache memory, but it would only be useful if the latency is lower than accessing the kernel space memory.

Regarding the analysis, a loopback execution has been performed between all the different considered memories, which basically moves consecutive data from one location to another. The data type used has been integers, and to better observe the time variation it has been executed with a range from 1 to 16K elements. Besides, the execution time of the data transfers analyzed has been obtained by dividing the total memory copy cycles (retrieved using hardware instrumentation) by the frequency, which has been 250 MHz.

Regarding the available memories, the considered ones are as follows:

- Kernel space memory, also referred to as CPU interface from the kernel accelerator point of view.
- DDR3 memory on the PL side, also referred to as Mem PL.
- Private internal BRAM.
- Shared external BRAM. This particular memory has at maximum two ports and can be accessed from the kernel accelerator by two ways: directly using a BRAM port, or using a BRAM controller, which on one side has a S_AXI port and on the other side a BRAM port.

In order to analyze the different memory transfers, it is better to consider real scenarios where using other memory types can enhance the performance. However, as mentioned previously, it is important to keep in mind that the kernel accelerator will always use its own private internal BRAMs to execute the accelerated task.

4.3.1 Copies from the kernel space memory

Once the data is moved from user to kernel space memory it is accessible to all the accelerator kernels, which in the original version copy it directly to the internal BRAM. Figure 8 shows the rest of the possibilities:

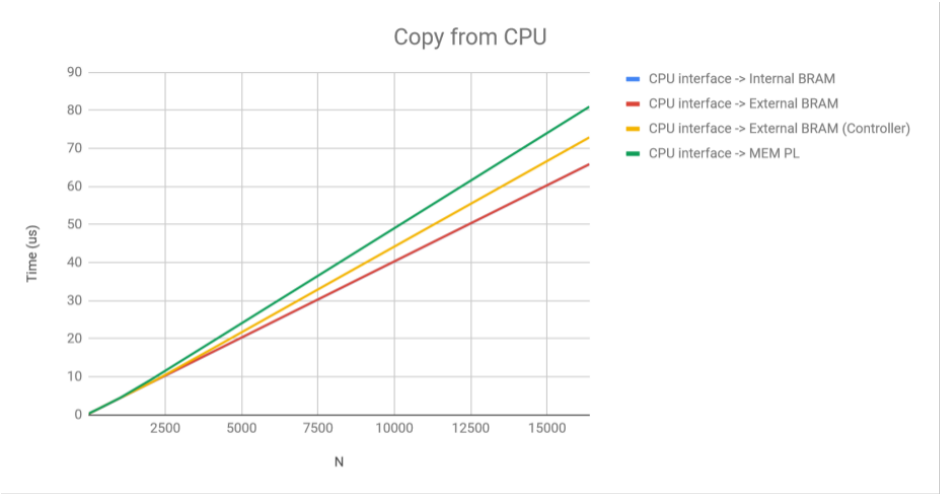

Figure 8: Copies from the kernel space memory to the rest of memories. Note: the CPU interface $\rightarrow$ Internal BRAM case is equal than the CPU interface $\rightarrow$ External BRAM case.

First of all, it is clear that the PL memory is the slowest one. Secondly, using a BRAM controller increases the access latency of the external BRAM. Finally, copying the data directly to the internal or external BRAM takes the same time, so it could be considered copying the data to the external BRAM and then from there to the kernel accelerators, depending on this last transfer latency. However, since the external BRAM has only two ports, one should be accessed directly from the kernel space memory, and the other should be accessed with a BRAM controller and an interconnection network should be placed when there were more than one kernel accelerator. Figure 9 shows graphically this situation.

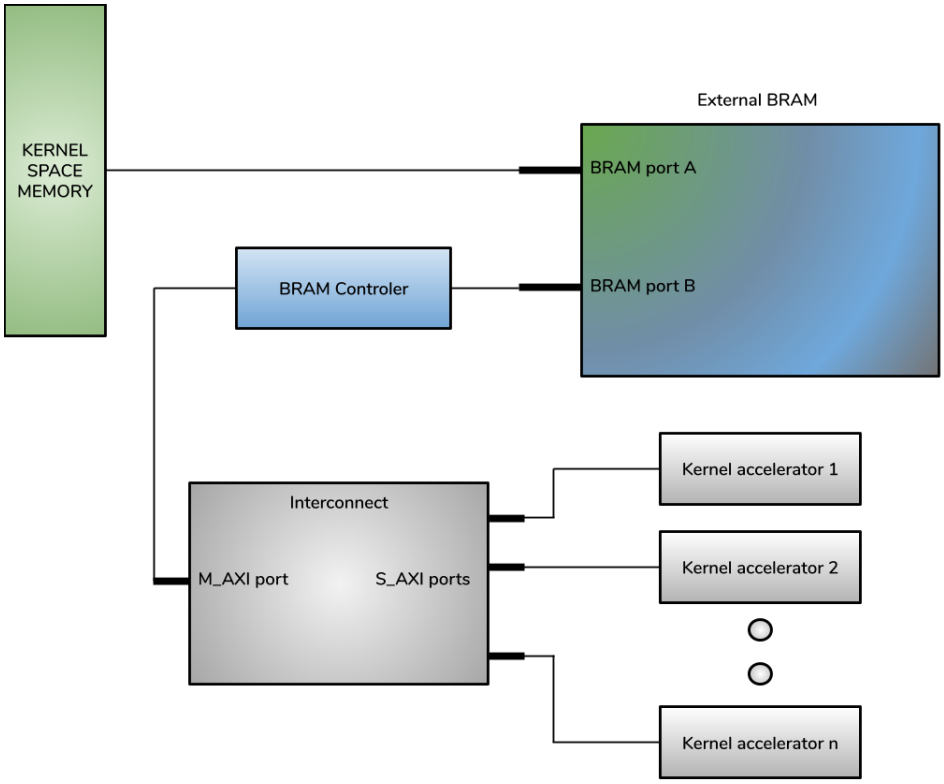

Figure 9: Simplified diagram when an external BRAM is used between kernel space memory and different kernel accelerators.

4.3.2 Copy Ins to the internal BRAM

Similar to the previous case, in this one the analysis is performed from the internal BRAM point of view. Figure 10 shows the different possibilities.

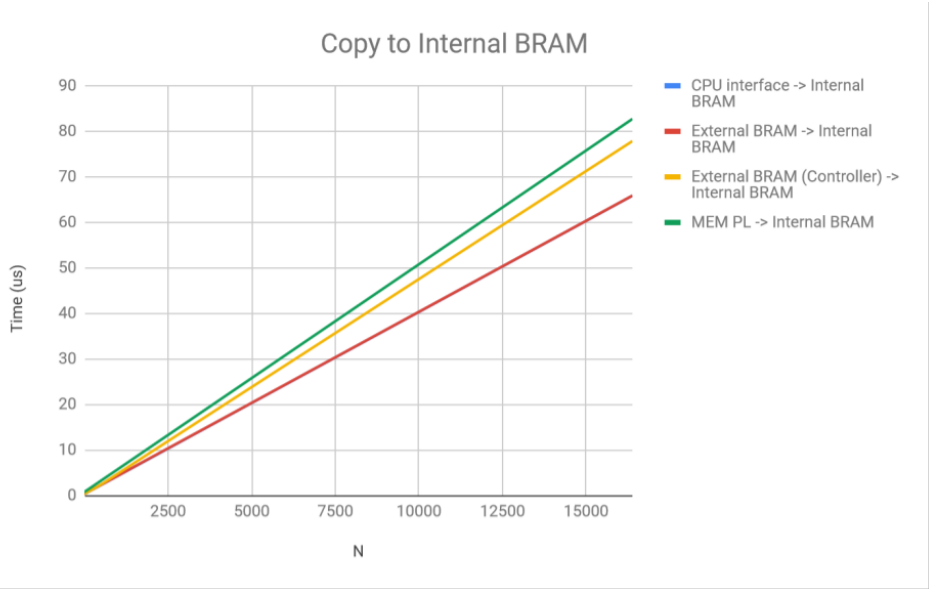

Figure 10: Copies to the internal BRAM of the kernel accelerator from the rest of memories. Note: the CPU interface → Internal BRAM case is equal than the External BRAM → Internal BRAM case.

The results are similar to the Figure 8: the PL memory is the slowest one, followed by the external BRAM when using a BRAM controller, and copying from the CPU interface is equal than directly from the external BRAM.

Considering this, the idea of copying data from the kernel space memory to a shared BRAM and then to the kernel accelerators is discarded. The shared BRAM provides more memory space, but it does not improve the performance, as for example a cache does. Moreover, an interconnect network and a BRAM controller must be placed in the block diagram, which increases the area utilization.

4.3.3 Copy Outs from the internal BRAM

Having analyzed the input copies to the accelerators, what remains is evaluating the other direction, the output copies from the accelerators. Apparently, the results should be the same in both directions, but the analysis does not reflect this behaviour. Figure 11 shows the different latencies when copying from the internal BRAM:

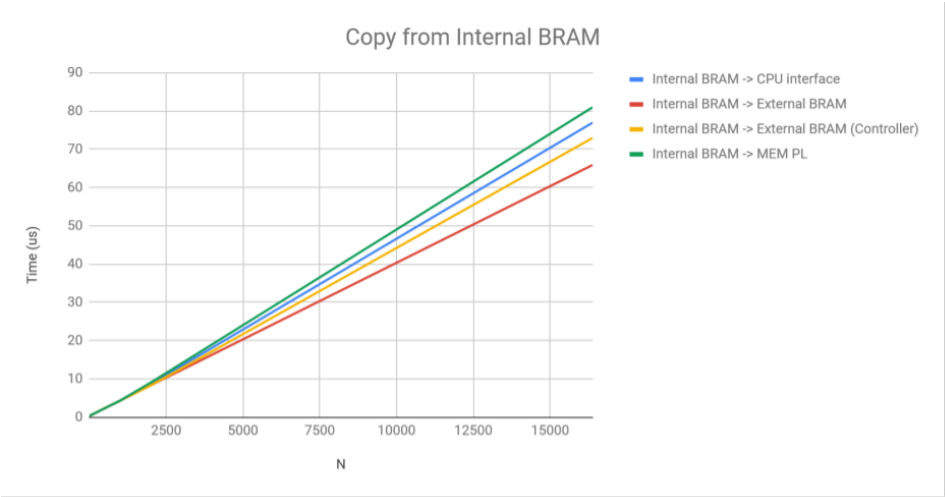

Figure 11: Copies from the internal BRAM of the kernel accelerator to the rest of memories.

While the PL memory and the directly external BRAM accesses remain the same as before, there are variations in the kernel memory space and the external BRAM with BRAM controller ones. The first one takes more time to write than to read, possibly due to a specific memory management of the DRAM; contrarily, the second one takes more time to read than to write, and it is also faster than accessing the kernel memory space.

Due to this, an option could be performing the output copies of the kernel accelerators to a shared BRAM, with or without a BRAM controller. Apart from being faster than writing to kernel space memory, it could be used as an intermediate memory region from where other kernel accelerators could read input data. However, considering Figures 9 and 10, when there were more than two kernel accelerators accessing this shared BRAM, a BRAM controller with an interconnect network should be placed, which would increase the latency and therefore not enhancement in the performance.

To sum up, after this memory transfers analysis there are some conclusions that have been considered for the mechanisms presented in this project. First of all, the idea of using shared BRAMs among the kernel accelerators as caches is discarded; even though they provide an intermediate space to place data that can be used by different kernel accelerators, the latency is almost equal than accessing the kernel space memory and the utilization area is increased. Secondly, accessing directly an external BRAM is faster than with a BRAM controller. Similarly, reading from CPU is faster than

writing to it. Lastly, the PL memory has high latency; however, it could be useful in some applications since it is large and capable of storing more data than in the kernel memory space.

Finally, this analysis has been extended to experiment with the port widths of the different memories. While the kernel space memory is limited to 32 bits, BRAMs and the PL memory can be expanded up to 1024 and 64 bits, respectively. Figure 12 shows the latency variations when using these bit widths with respect to the original ones:

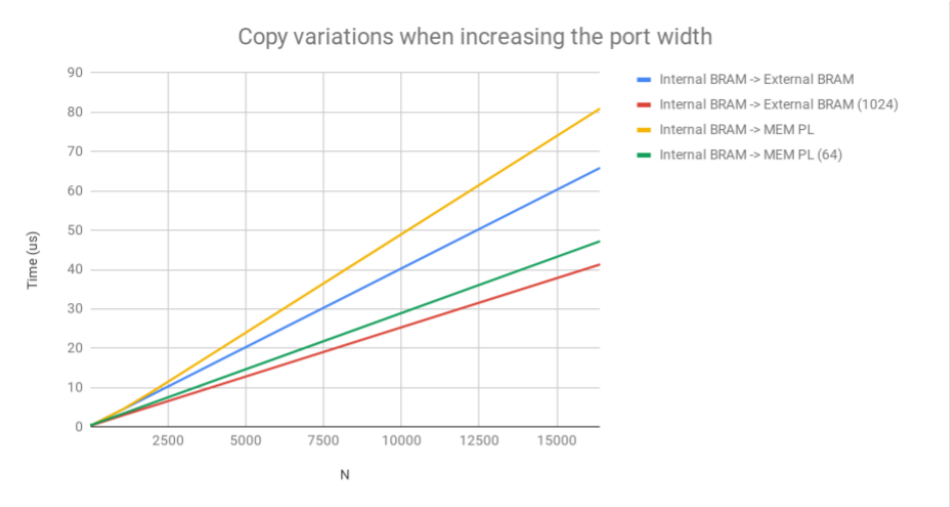

Figure 12: Copy variations with respect to the original ones from internal BRAM to external BRAM and PL memory when expanding the port widths.

However, the data must be processed correctly to read and write memory transfers widths different than the data type one, in this case integer (32 bits). Moreover, the performance is maximized when the read and write ports have the same bit width. Nevertheless, this project will not focus on these specific details, since they can vary depending on the application and the purpose of OmpSs@FPGA is to provide a generic FPGA task offloading system, but they can be considered for later enhancements.

Chapter 5

Data reuse support design

In this chapter the different techniques explored in order to try to reuse data and their designs are presented. There are four, which have been named as Only-software, Only-hardware, Relaxed software-hardware and Aggressive software-hardware.

5.1 Only-software

This method was initially developed to verify in a rapid manner how the copy flags of the kernel accelerators could be managed to reuse data. Since the modifications are only done in the software part, which includes Nanos++ and xTasks, the hardware does not have to be reconfigured and therefore there is a significant time saving. However, this method focused on the input copies and was developed for FPGAs that have only one accelerator, since the execution times obtained were too unsatisfactory to put more effort to implement communication mechanisms between FPGA accelerators, even though it was useful as a proof of concept.

To handle the FPGA tasks offloading, Nanos++ separates the process in two steps. First, it determines if the data needed was copied previously to the kernel space memory, which would imply that the kernel memory region is mapped to a user memory region. If not, Nanos++ performs the transfer using the *_copyIn* function, while at the same time maps that specific kernel region with the corresponding user region. Besides, a mapped kernel region can be unmapped (also referred to as invalidated) if another device task modifies its user region, or if a taskwait is encountered during the execution. Second, it prepares and submits the FPGA tasks using the *createTask*, *setTaskArg* and *submitTask* functions, which at the same time

call the xTasks library. Figure 13 shows a flow chart depicting this FPGA task submission.

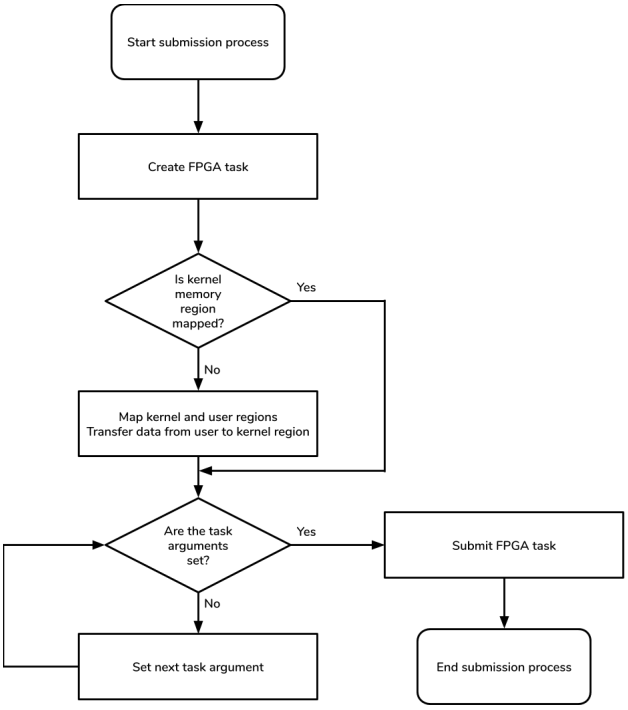

Figure 13: Submission process of an FPGA task.

Taking into account this process submission, the strategy followed has been to check what data blocks the kernel accelerator has and their mapping validity with the user memory space before submitting the task. To this end, Nanos++ must keep the kernel accelerator copies information, and manage the copy In flags when setting the arguments task. Figure 14 shows the strategy flow chart:

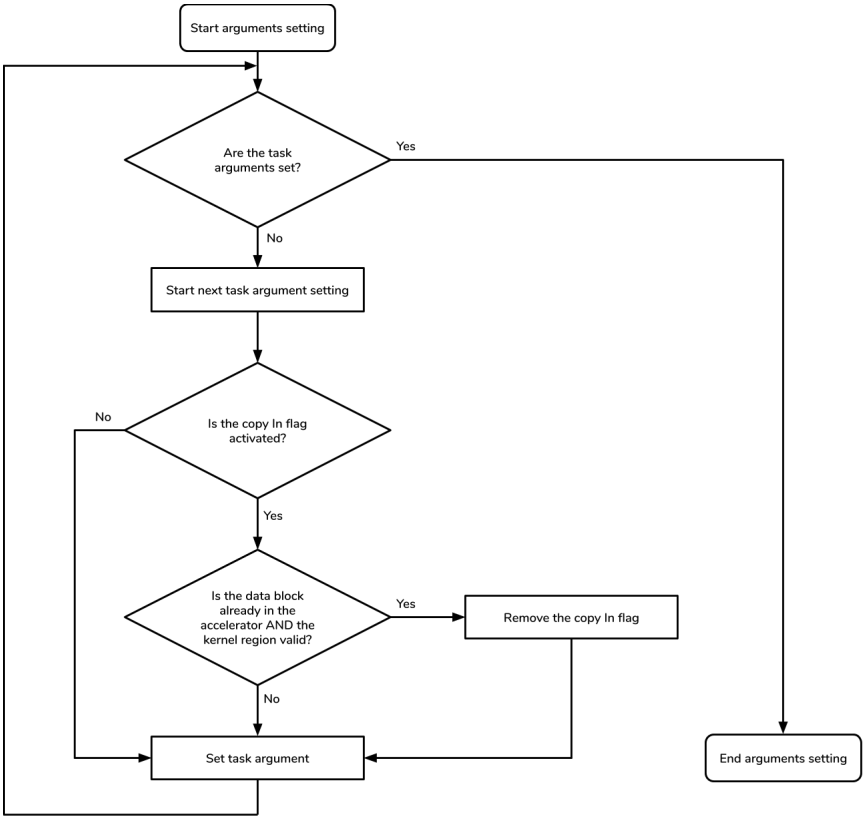

Figure 14: Only-software flow chart design.

As it can be observed, in this design the data reuse logic is controlled entirely by the software part. Regarding the hardware part, it does not need to be modified and will continue receiving the ready tasks as before, but some copy In flags may be deactivated.

5.2 Only-hardware

In this design the strategy followed was the opposite than the previous one. Instead of modifying the Nanos++ runtime, it is kept unchanged and the modifications are done in the hardware part, concretely in the ReadyTaskManager.

For each kernel accelerator, its tasks that are ready for being executed (because their dependencies have been fulfilled) are located in the readyQueue BRAM. Their execution order will be determined by their submission order,

and it will not be changed at any moment. Because of this, it is possible to know if two tasks that will be executed consecutively in the same kernel accelerator will share any of their data blocks, and consequently deactivate their copy flags.

However, there are three considerations that have to be taken into account in this design. The first one is that, in order to deactivate the copy flags, two consecutive tasks of the same kernel accelerator must be ready in the readyQueue BRAM when comparing them; because of this, it also depends on the submission velocity from Nanos++. The second one is that a task must compare its data blocks against the next task, and not the previous task. Otherwise, it is not possible to determine if any shared data blocks were modified by another kernel accelerator or device between their submission times. Lastly, the third one is that in this design the only copy flags that may be deactivated are the input ones. This is due to a dependency constraint, since two tasks can not be ready at the same time if they share any equal output data block because it would imply a WaW dependency.

Figure 15 shows an execution flow example of this design with one kernel accelerator which have block A as input data. The data block comparisons of the first FPGA task are done before the second FPGA task is ready, and therefore the block A will be copied in twice when it could have been avoided.

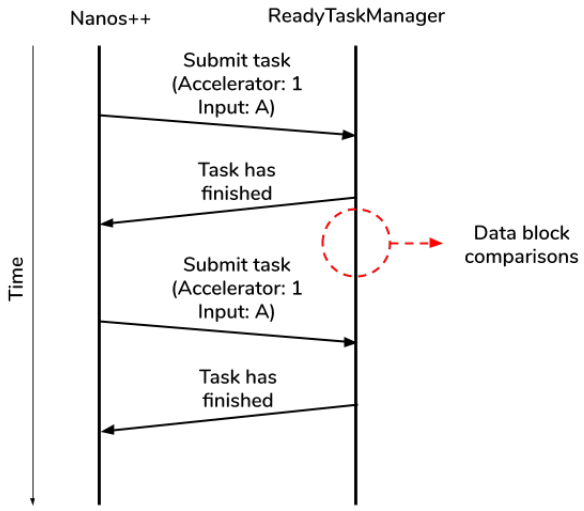

Figure 15: Execution flow example with the Only-hardware design where the data block comparisons is performed before the next task is ready.

Figure 16 shows an execution flow example of this design when a kernel accelerator and the SMP have a data dependency on the block A. In this particular situation, if the data block comparisons were done with the previous task, the second FPGA task would have seen that the first FPGA task copied in the block A, and therefore it would have deactivated its copy In flag. However, this behaviour would have been incorrect since the data of the block A was updated by an SMP task.

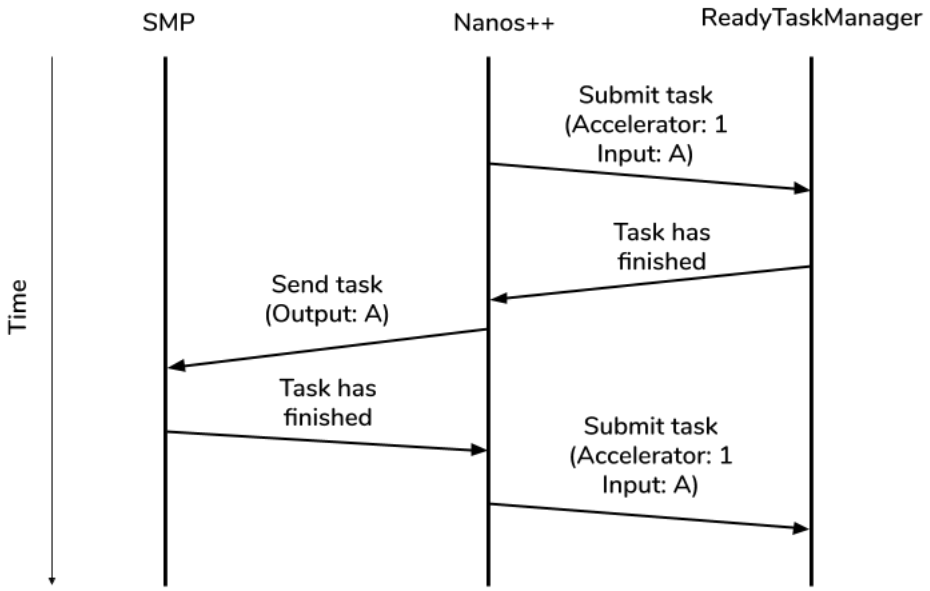

Figure 16: Execution flow example with the Only-hardware design where there are data dependencies between an FPGA and an SMP task.

Finally, Figure 17 shows graphically the steps explained in this process.

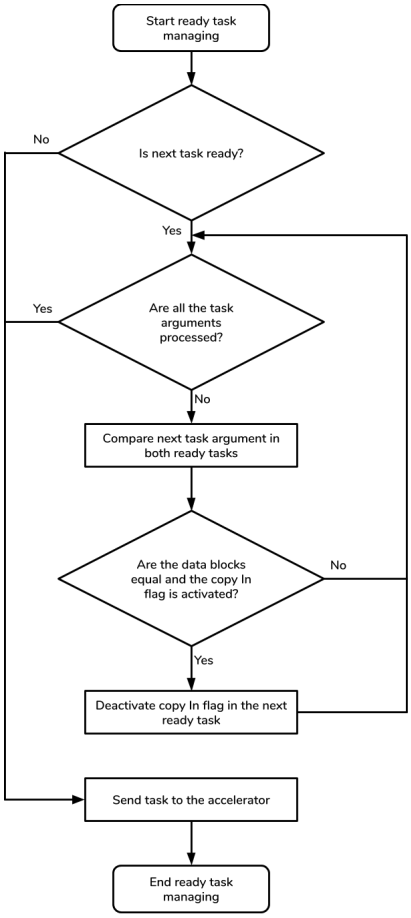

Figure 17: Only-software flow chart design.

5.3 Relaxed software-hardware

After designing the two previous methods, their deficiencies were analyzed in order to think of another technique that could mitigate them. On the one hand, implementing all the data reuse logic on the software part implies an overhead that decreases the overall performance. On the other hand, even though the hardware part can manage the copy flags in a more efficient way, it also depends on the readiness of the tasks at the moment of comparing them. Because of this, an hybrid technique has been designed to try to reduce these constraints.

Regarding the data blocks present in the kernel accelerators, the software

and the hardware part act like separate black boxes unable to indicate their states. While Nanos++ does not know their execution flow once they are submitted to the readyQueue BRAM, the hardware can not determine if the shared input data blocks of two consecutive ready tasks of the same kernel accelerator contain the same data, if they were not ready at the same time. Because of this, a new shared BRAM between Nanos++ and the hardware part has been designed, named dataIn.

The dataIn BRAM contains the information of the data blocks present at any moment for each kernel accelerator. This information is basically the data block address and its state, which in this design can be valid or invalid, and can be accessed by both Nanos++ and the hardware part. As a result, a VI protocol has been designed whose state transitions depend on the read/write requests performed during the execution flow by the different devices of the system. Figure 18 depicts the finite-state machine of the data blocks present on the kernel accelerators.

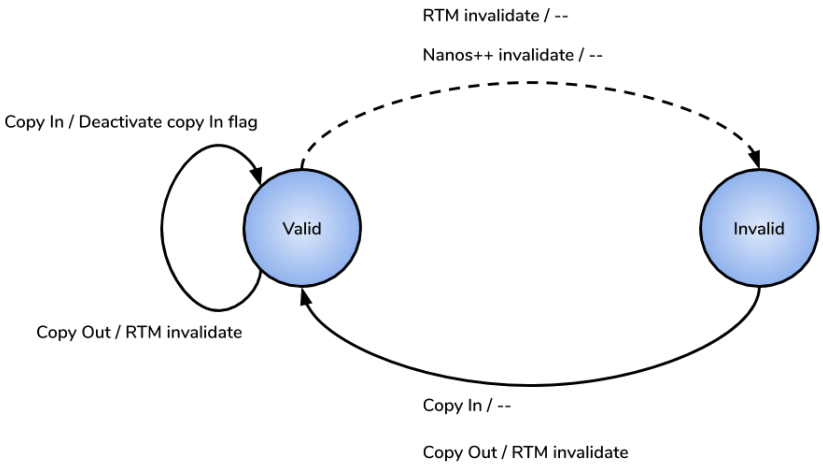

Figure 18: Finite-state machine of the data blocks present on the kernel accelerators.

As it can be noticed, there are two types of invalidations, which are performed when an output block address of a task matches with a valid block address. While the *Kernel invalidate* is performed by the ReadyTaskManager and considers only FPGA tasks, the *Nanos++ invalidate* is performed by Nanos++ and considers tasks from other devices, such as SMP or GPU. This last one can be determined when Nanos++ transfers data from user to

kernel space memory, since it is updating the data accessible by the kernel accelerators, and additionally is able to detect implicit invalidations due to taskWaits. Besides, it is also important to consider that a valid block that is being substituted by a new one must be evicted from the dataIn.

Considering this new dataIn BRAM, this technique aims to apply an enhanced *Only-hardware* strategy. Instead of comparing the current task blocks with the next task ones, the ReadyTaskManager examines the dataIn BRAM regarding the kernel accelerator that will execute the task, and deactivate its copy In flags accordingly. Figure 19 shows the resulting flow chart of the ReadyTaskManager in this design.

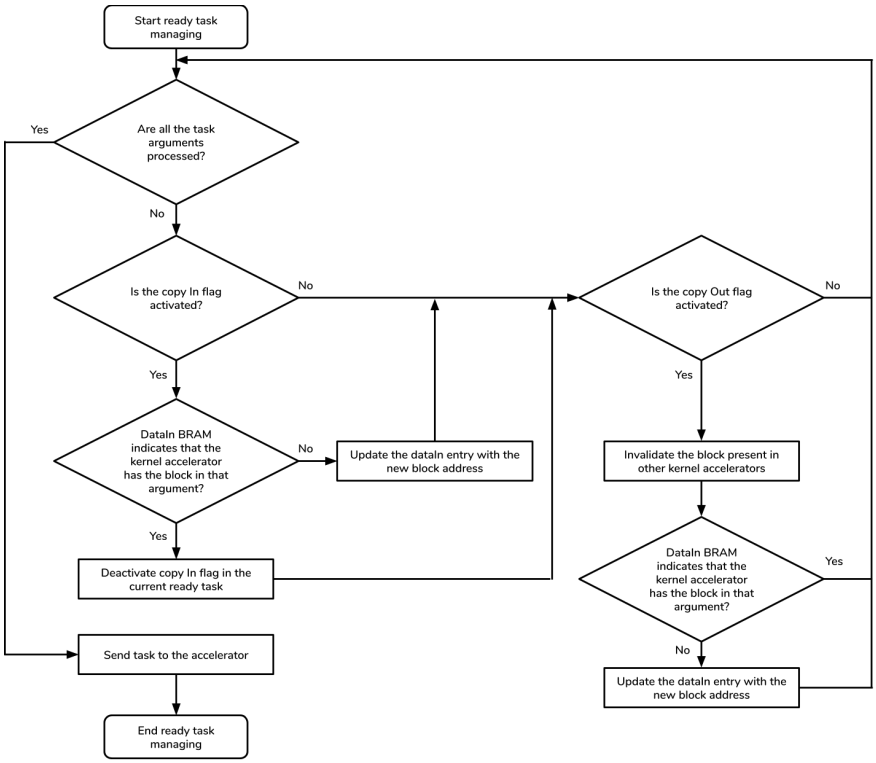

Figure 19: Relaxed software-hardware flow chart design of the ReadyTaskManager.

Figure 20 shows an execution flow example with this design where two kernel accelerators have a data dependency on the block A; however, both kernel accelerators do not interfere with each other executions. The first task of the kernel accelerator #1 causes the block A to be copied in, and

the corresponding dataIn entry is updated to the V state. After some time (depicted with short diagonal lines) a new task to the kernel accelerator #1 arrives, and since that dataIn entry has not been modified, the copyIn flag can be deactivated because a valid block A is present on it. Lastly, a task to the kernel accelerator #2 arrives, which provokes its block A entry in the dataIn to be updated to the V state, and the invalidation of the kernel accelerator #1 one.

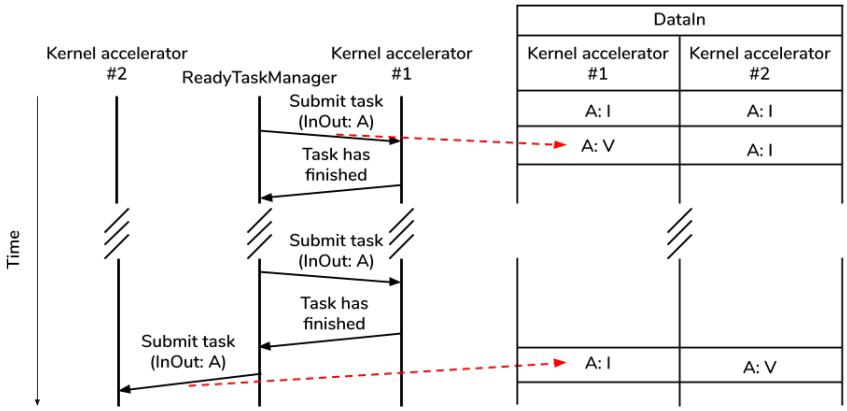

Figure 20: Execution flow example with the Relaxed software-hardware design where there are data dependencies between two FPGA tasks.

In contrast, Figure 21 shows an execution flow example with this design where there is only one kernel accelerator but Nanos++ performs invalidations too. The first task of the kernel accelerator #1 causes the block A to be copied in, and the corresponding dataIn entry is updated to the V state. However, Nanos++ invalidate that entry later, due to another device has updated the data block or a taskwait has been encountered. As a consequence, the second task of the kernel accelerator #1 must copy in the block A because the dataIn entry was in the I state.

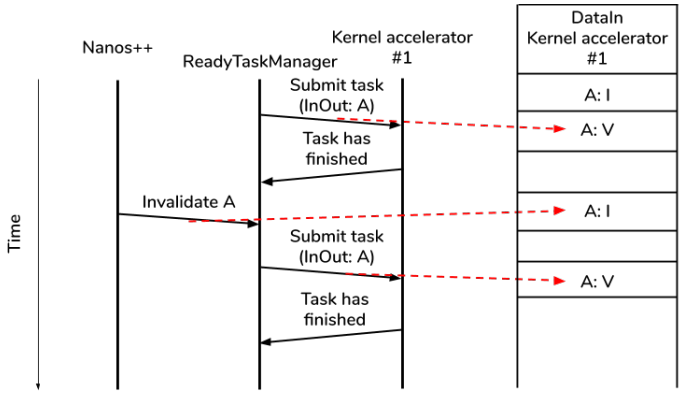

Figure 21: Execution flow example with the Relaxed software-hardware design where Nanos++ invalidates a block present in a kernel accelerator.

5.4 Aggressive software-hardware

Finally, a method was designed in order to avoid the copy out of data from the kernel accelerators when it is possible. This design extends the *Relaxed software-hardware* one, and implies that a kernel accelerator may have a private updated data block.

In the previous designs there were only two memory spaces where the last updated data could be, the user and the kernel ones. However, if the kernel accelerators are capable of avoiding copying out data, a new memory space is included for each kernel accelerator. Because of this, apart from invalidating valid blocks, in this design there are also petitions of requesting data to the kernel accelerators.

To this end, a Modified-Shared-Invalid (MSI) protocol has been designed for the data blocks present in the kernel accelerators. It is similar to the VI protocol presented before, but with the particularity that the M state indicates that a kernel accelerator has a private updated data block. Figures 22 and 23 depict the finite-state machine of the data blocks present on the kernel accelerators considering the copy flags and the events from other kernel accelerators and Nanos++, respectively.

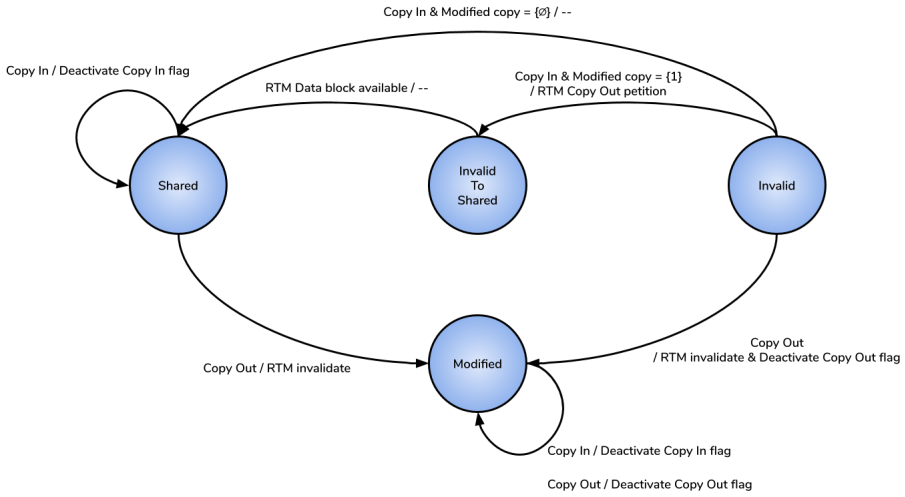

Figure 22: Finite-state machine of the data blocks present on the kernel accelerators considering the copy flags.

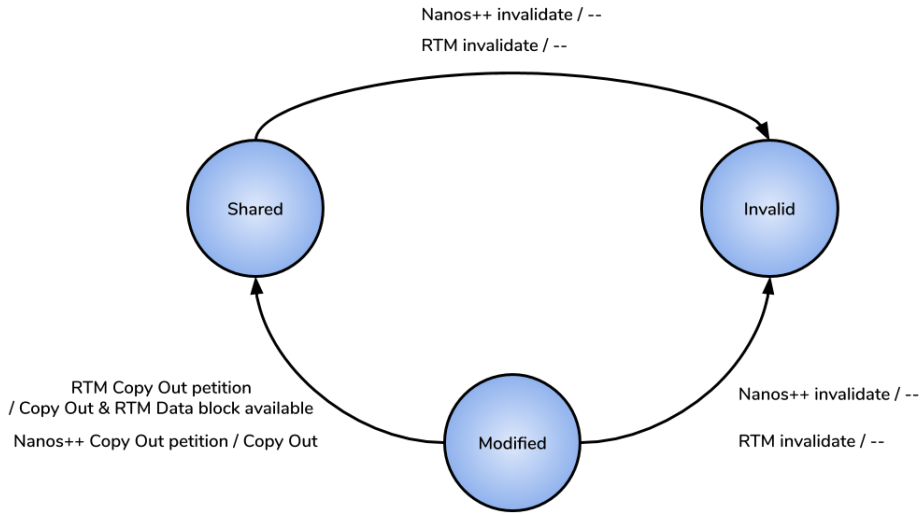

Figure 23: Finite-state machine of the data blocks present on the kernel accelerators considering the events from other kernel accelerators and Nanos++.

As it can be seen, there are several modifications on the finite-state machine compared to the VI protocol. The most relevant is the one related

to copying out data, since the copy Out flags are used only to determine if a block will pass to the M state, and then they are deactivated. Therefore, the data is copied out from the kernel accelerators explicitly only when another kernel accelerator or Nanos++ request it. Besides, the requested data will be placed in the kernel memory space as usual, since it is accessible to all the components involved.

However, this new data flow communication between kernel accelerators will not be determined by a Nanos++ task, since it is managed in the ReadyTaskManager. For that reason it has been designed a mechanism that, employing the dataIn BRAM, allows a kernel accelerator to request to another one the copying out of a data block. Consequently, the demanding kernel accelerator will not be able to start its execution until its input data is accessible. Figure 24 shows an example of this situation.

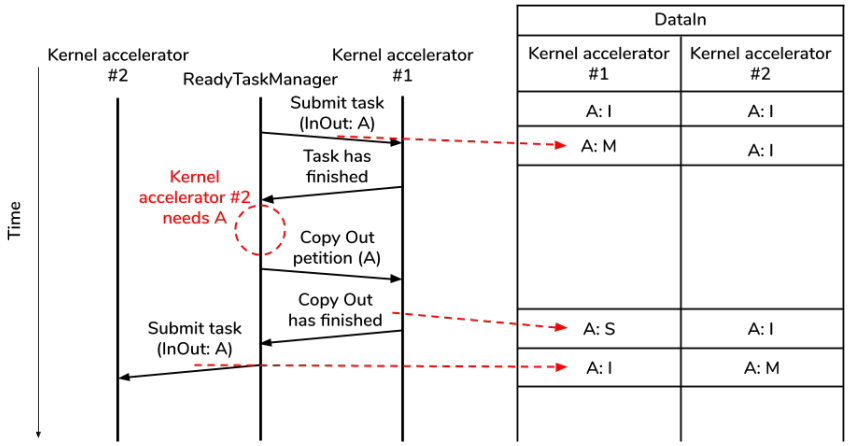

Figure 24: Execution flow example with the Aggressive software-hardware design where two accelerators have a data dependency on block A and a data request is performed.

To handle this new situation, the execution flow of the readyTaskManager has been modified to force the copy out of requested data from the kernel accelerators. Figure 25 shows the flow chart design of the ReadyTaskManager, where it can be seen that before submitting a task it has to be analyzed if another kernel accelerator or Nanos++ requested a modified data. Moreover, it also has to be checked if the data dependencies with other kernel accelerators have been resolved; otherwise, the kernel accelerator must wait until they are resolved.

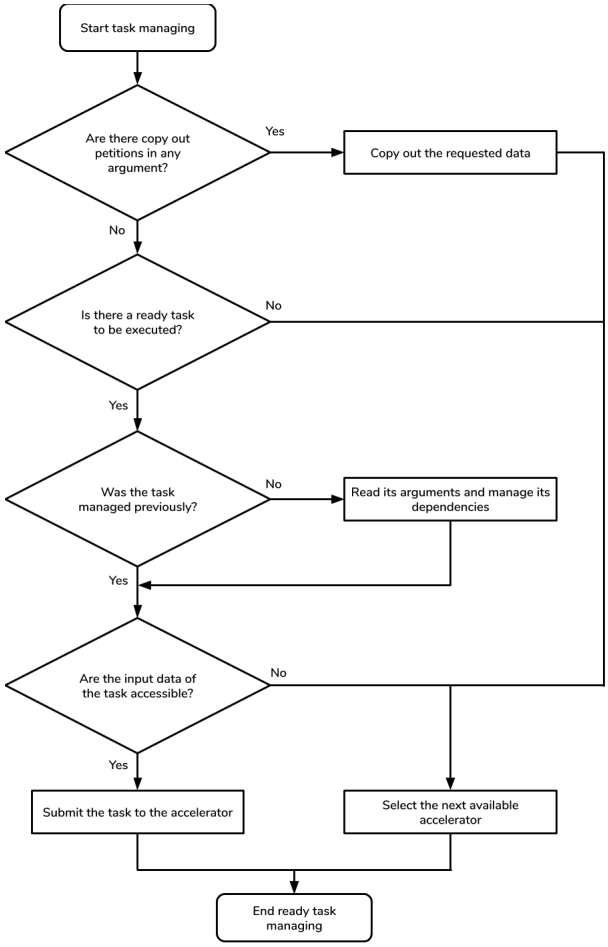

Figure 25: Aggressive software-hardware flow chart design of the ReadyTaskManager.

Figure 26 shows the flow chart design of the ReadyTaskManager when managing the tasks. As mentioned before, its peculiarity is that it has to be explored if any kernel accelerator has a modified copy of the input data of the task, and request it. Similar to the *Relaxed software-hardware* design, a valid block that is being substituted by a new one must be evicted from the dataIn, but if its state was M it must be copied out before. Besides, it has to be noticed that the copy Out flag is only used for managing the block states, but deactivated when submitting the task.

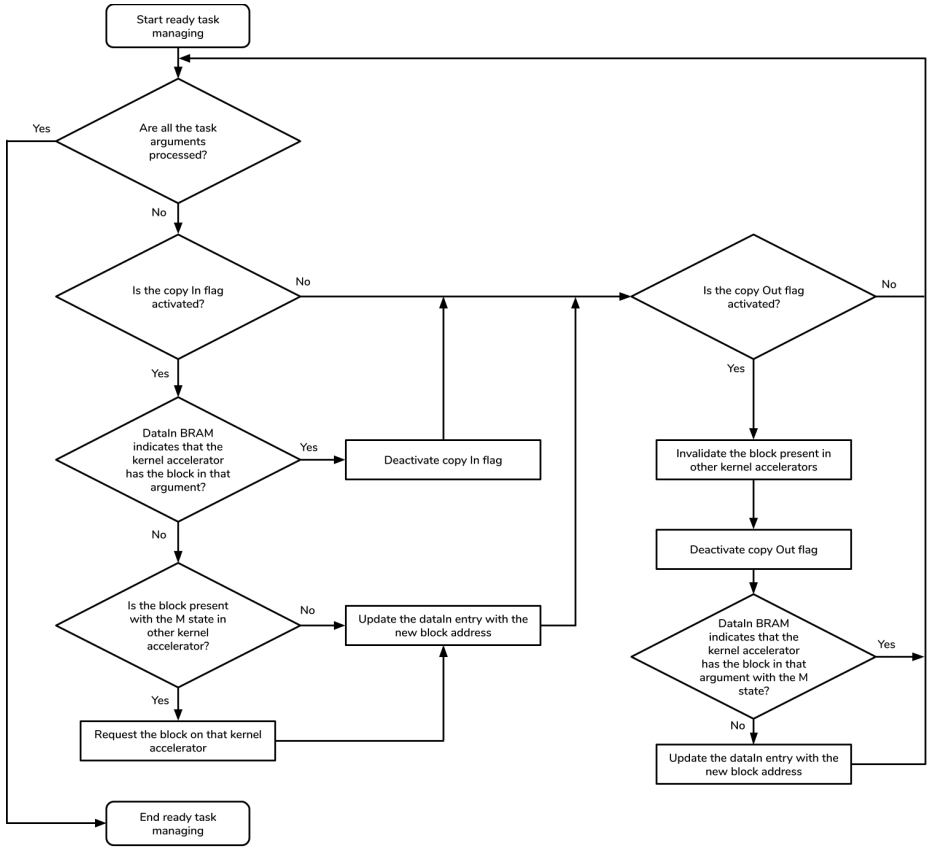

Figure 26: Aggressive software-hardware flow chart design for managing the tasks in the ReadyTaskManager.

Chapter 6

Implementation details

Once the different techniques designs have been presented, this chapter will explain their implementations details, as well as the encountered difficulties.

6.1 Only-software

Before starting with the implementation of this design, it is important to clarify some relevant aspects of the FPGA tasks execution flow. In OmpSs@FPGA several threads can be managing concurrently the pool of ready FPGA tasks that have to be created, processed and submitted in order to accelerate the offloading to the FPGA. This implies that any access to a shared variable must be serialized to ensure the memory consistency. Besides, it is guaranteed that a specific FPGA task will be managed by the same thread during all the submission process.

As mentioned before, it is necessary to know the data blocks present in a FPGA kernel accelerator to implement this design. To this end, the Nanos++ class that instantiates the FPGA accelerators has been expanded with three additional fields, which are:

- currentBlocks, to keep the current blocks present in the kernel accelerator. It maps each argument ID with its data block address.
- currentTaskArgs, to keep the arguments information of a task. It maps, for each <task, argument ID>, the argument information (copy flags and data block address).
- pendingCopies, a set to indicate that a kernel region has been updated and the input copy must be performed.

At this point, Nanos++ is able to retrieve the kernel accelerator copies information to manage the copy flags. However, the accesses to these additional data fields must be serialized since more than one thread can be submitting a task to the same accelerator. The resulting involved functions after including the input copies logic are shown in Algorithms 4, 5 and 6.

Algorithm 4 Only-software _copyIn implementation

1: Acquire lock
2: Add the block address to the *pendingCopies* set
3: Copy the data from user to kernel memory space
4: Release lock

Algorithm 5 Only-software setTaskArg implementation

1: Read the task ID
2: Read the argument data address
3: Read the argument copy flags
4: Read the argument ID
5: Acquire lock
6: Insert a new entry with <<task ID, argument ID>, argument information> to *currentTaskArgs*.
7: Release lock

The most notorious drawback in this design is the complete serialization of the *submitTask* function. The submission order of the ready tasks is determined dynamically at execution time, and different threads may be modifying the *currentBlocks* shared variable concurrently. Therefore, since the decision of removing the copy In flag is based on this variable, which indicates the data blocks that will be present on the kernel accelerator when the ready task has to be executed, the different submits can not interfere with each other.

6.2 Only-hardware

To implement this design, only the ReadyTaskManager needs to be modified. Since it manages the ready tasks one by one in the submission order, there is no need for including additional locks, like in the previous design. However, apart from sharing input data blocks, two consecutive tasks must be ready when the ReadyTaskManager compares their arguments in order

Algorithm 6 Only-software submitTask implementation

```
1: Get task ID
2: Acquire lock
3: for all Arguments to process do
4:    Get the argument information from currentTaskArgs
5:    if Argument has the copy In flag then
6:       if Argument data address is in pendingCopies then
7:          Update currentBlocks[argument ID] with the argument data address
8:          Remove the argument data address from pendingCopies
9:       else if currentBlocks[argument ID] is different than the argument data address then
10:         Update currentBlocks[argument ID] with the argument data address
11:      else
12:         Remove the copy In flag from the argument copy flags
13:      end if
14:   end if
15:   Set the task argument
16: end for
17: Submit the task
18: Release lock
```

to deactivate the input copy flags. Algorithm 7 shows the resulting ReadyTaskManager logic when implementing this design:

Algorithm 7 Only-hardware ReadyTaskManager implementation

```
1: for all Accelerators do
2:    acceleratorEntry ← 0
3: end for
4: acceleratorIndex ← 0
5: while true do
6:    if Task in ReadyQueue[acceleratorEntry[acceleratorIndex] is ready
      and accelerator is available then
7:       if Task was not read previously then
8:          Read task
9:       end if
10:      if Next task of the accelerator is ready then
11:         Read next task
12:         for all Task arguments do
13:            if Current task argument address == Next task argument address then
14:               Deactivate next task argument copy In flag
15:            end if
16:         end for
17:      end if
18:      Send task to the accelerator
19:      Mark the entry as free
20:      Mark the accelerator as busy
21:      acceleratorEntry[acceleratorIndex] + +
22:   else
23:      acceleratorIndex + +
24:   end if
25: end while
```

As it can be seen in the implementation pseudocode, a task is read from the ReadyQueue BRAM only once. This is an optimization technique applied to avoid reading twice those ready tasks that were read previously to compare it with the current task. To this end, double buffering is performed on an internal buffer that keeps, for each kernel accelerator, the current and the next ready task.

6.3 Relaxed software-hardware

This design requires several modifications which can be implemented with different approaches, depending on how the dataIn BRAM is accessed by the ReadytaskManager. Since there are situations where Nanos++ and the ReadytaskManager have to invalidate blocks present in kernel accelerators, the dataIn BRAM must be explored to find these blocks. Therefore, this exploration is an additional overhead that can influence the overall performance. Besides, the copy flags managing logic is essentially the same in all the approaches.

Regarding the ReadytaskManager, the initial implementation iterated the dataIn entries of each accelerator one by one to find blocks that had to be invalidated. This, logically, was inefficient and implied an overhead to the ReadyTaskManager that reduced its performance. To overcome this issue, a Pearson hash function[33] has been used to invalidate the corresponding data blocks in a faster way, since it is simple and designed for fast executions

The Pearson hash function associates a given block address with an index to an array named *hashArray*. Since two different block addresses may have the same value returned by the Pearson hash function, another array named *synonymsArray* concatenates them. Moreover, it also keeps their block addresses to differentiate them. This last array stores the necessary information to easily retrieve the dataIn entries (acceleratorID and argumentID), and each of the *hashArray* entries points to the first element of the concatenated blocks.

When the dataIn is updated, implicitly the *synonymsArray* must be updated too. If a data block of a kernel accelerator is invalidated or substituted by another one, it has to be deleted from the *synonymsArray*, and its concatenated blocks regrouped. Contrarily, if a kernel accelerator copies in a new block, it has to be inserted to the *synonymsArray*, and it becomes the first element on its concatenated list.

Regarding Nanos++, initially it also iterated the dataIn entries of each accelerator one by one, and directly modified the corresponding blocks states to Invalid. Similar than in the ReadytaskManager, this was inefficient and decreased the task submission speed of Nanos++. Moreover, these Nanos++ invalidations will lead to errors when using the Pearson hash in the ReadyTaskManager, since the *synonymsArray* would not be updated. To solve this problem, a new BRAM named *invalidationQueue* has been included.

The *invalidationQueue* BRAM is shared by Nanos++ and the ReadytaskManager and used to perform Nanos++ invalidations. When Nanos++

has to invalidate a block, it updates an *invalidationQueue* entry with the block address, which is read by the ReadytaskManager an appropriately invalidated in the dataIn. Algorithms 8, 9 and 10 show the resulting *_copyIn* Nanos++ function, the *invalidationQueue* managing in the ReadytaskManager, and the copy flags managing, respectively.

Algorithm 8 Relaxed software-hardware _copyIn implementation

1: Search for a free *invalidationQueue* entry
2: Indicate the kernel memory address where the new data will be copied on that entry
3: Mark the entry as valid
4: Copy the data from user to kernel memory space

Algorithm 9 Relaxed software-hardware invalidations managing from the ReadyTaskManager

1: $invalidationQueue\_index \leftarrow 0$
2: **while** true **do**
3: **if** $invalidationQueue[invalidationQueue\_index]$ is valid **then**
4: Read the invalidation address
5: Search for the first element of the concatenated blocks in *synonymsArray* that map with the Pearson hash function
6: **for all** The elements of the concatenated list that have the invalidation address **do**
7: $AcceleratorID \leftarrow element.accID$
8: $ArgumentID \leftarrow element.argID$
9: $dataIn[AcceleratorID][ArgumentsID].state \leftarrow Invalid$
10: Delete the element from the *synonymsArray*
11: **end for**
12: $invalidationQueue_index \leftarrow invalidationQueue_index + 1$
13: **end if**
14: **end while**

6.4 Aggressive software-hardware

Finally, this design has required modifications in the readyTaskManager and Nanos++. The first one has been implemented as an extension of the *Relaxed software-hardware* implementation, and the second one has required

Algorithm 10 Relaxed software-hardware ReadyTaskManager implementation

```
1: for all Accelerators do
2:    acceleratorEntry ← 0
3: end for
4: acceleratorIndex ← 0
5: while true do
6:    if Task in ReadyQueue[acceleratorEntry[acceleratorIndex] is ready
      and accelerator is available then
7:       Read task
8:       for all Task arguments do
9:          if Task argument has copy In flag activated then
10:            if dataIn[acceleratorIndex][argument] state is Valid then
11:               Deactivate task argument copy In flag
12:            else
13:               Insert the task argument address with acceleratorIndex and
                  argumentID to the synonymsArray
14:            end if
15:         end if
16:         if Task argument has copy Out flag activated then
17:            for all Elements in the synonymsArray that have the task ar-
               gument address and are present in a different kernel accelerator
               do
18:               Delete the element from the synonymsArray
19:            end for
20:         end if
21:      end for
22:      Send task to the accelerator
23:      Mark the entry as free
24:      Mark the accelerator as busy
25:      acceleratorEntry[acceleratorIndex] + +
26:   else
27:      acceleratorIndex + +
28:   end if
29: end while
```

to implement a function to ask for private modified data blocks of the kernel accelerators.

Regarding the readyTaskManager, there are three relevant parts that have changed with respect to the *Relaxed software-hardware* one. The first one is the managing of the available accelerators, which has been extended with more cases to take into account. In previous designs the available kernel accelerators were only considered when they had a task ready to be executed. However, since in this design a kernel accelerator can receive copy outs petitions, they are also considered even if there is not a ready task. Consequently, a ready task can be postponed if there is a copy out petition. Furthermore, a task can not be executed until the possible data dependencies with other accelerators are resolved. Algorithm 11 shows this accelerators managing.

The second one is how the copy flags of the ready tasks are managed. Since in this design a kernel accelerator may substitute a data block that is in the Modified state, first of all it has to check its own data blocks present at that moment, as it is shown in Algorithm 12. Besides, when comparing with another kernel accelerators, apart from taking into account the invalidations, in this design the requests to other kernel accelerators for a modified data block have to be considered too. Algorithm 13 shows this process.

Finally, a new function named submitCopyOutTask has been added in order to perform the data requests to kernel accelerators. It receives an array with the output copy flags activated in the desired arguments, and sends them to the kernel accelerator target. However, the task header is set to zero because it does not come from a Nanos++ task and therefore it does not have any task information, which provokes that the output copies can not be instrumented and visualized. Algorithm 14 shows the pseudocode of this function.

Regarding Nanos++, a new function named xtasksCopyOut has been added in order to ask for modified data blocks present in the kernel accelerators. This function is called when Nanos++ transfers data from kernel to user space memory, and it also uses the dataIn BRAM to make the petitions. However, it has the drawback that the dataIn must be explored block by block until the memory space address coincides with a block address, which is not always. Besides, once it finds the modified block, it can not continue until the data is in the kernel space memory. Algorithm 15 shows the pseudocode of this function.

Algorithm 11 Aggressive software-hardware ReadyTaskManager implementation

```
1: for all Accelerators do
2:   acceleratorEntry ← 0
3: end for
4: acceleratorIndex ← 0
5: while true do
6:   if Accelerator is available then
7:     if There are copy out requests then
8:       outRequests ← ∅
9:       for all Requested arguments do
10:        outRequests+ =< argumentID, blockaddress >
11:      end for
12:      submitCopyOutTask(outRequests)
13:    else if Task in ReadyQueue[acceleratorEntry[acceleratorIndex] is ready then
14:      task ← ReadyQueue[acceleratorEntry[acceleratorIndex]
15:      readArgsAndDependencies(task)
16:      if There are no data dependencies with other kernel accelerators then
17:        Send task to the accelerator
18:        Mark the entry as free
19:        Mark the accelerator as busy
20:        acceleratorEntry[acceleratorIndex] + +
21:      else
22:        acceleratorIndex + +
23:      end if
24:    else
25:      acceleratorIndex + +
26:    end if
27:  else
28:    acceleratorIndex + +
29:  end if
30: end while
```

Algorithm 12 readArgsAndDependencies Self

```
1: Read task
2: argsToProcess ← ∅
3: for all Task arguments do
4:    if The current address block in that argument is different than the
      new one and is not Invalid then
5:       if The current state block is Modified then
6:          Request the kernel accelerator to copy out the block
7:       else
8:          Delete from the synonymsArray the element that has the cur-
            rent address block in that argument and is present in this kernel
            accelerator
9:       end if
10:      argsToProcess+ = argumentID
11:   else if The current address block in that argument is equal than the
      new one and is not Invalid then
12:      if Task argument has copy In flag activated then
13:         Deactivate task argument copy In flag
14:      end if
15:      if Task argument has copy Out flag activated then
16:         if The current state block is Shared then
17:            Change the current state to Modified
18:            argsToProcess+ = argumentID
19:         end if
20:      end if
21:   else
22:      argsToProcess+ = argumentID
23:   end if
24:   Deactivate task argument copy Out flag
25: end for
```

Algorithm 13 readArgsAndDependencies Others

```
1: for all Arguments in argsToProcess do
2:   for all Elements in the synonymsArray that have the argument address and are present in a different kernel accelerator do
3:     AcceleratorID ← element.accID
4:     ArgumentID ← element.argID
5:     ArgumentState ← dataIn[AcceleratorID][ArgumentsID].state
6:     if ArgumentState is Shared then
7:       if Task argument has copy In flag activated then
8:         Delete the element from the synonymsArray
9:       end if
10:    else if ArgumentState is Modified then
11:      if Task argument has copy In flag activated then
12:        Request the kernel accelerator to copy out the block
13:        Update the dataIn entry state to Shared
14:        Insert the task argument address with acceleratorIndex and argumentID to the synonymsArray
15:      end if
16:      if Task argument has copy Out flag activated then
17:        if Task argument has copy In flag activated then
18:          Delete the element from the synonymsArray
19:        end if
20:        Update the dataIn entry state to Shared
21:        Insert the task argument address with acceleratorIndex and argumentID to the synonymsArray
22:      end if
23:    end if
24:  end for
25: end for
```

Algorithm 14 submitCopyOutTask

```
1: Send the task header as all 0s
2: for all Task arguments do
3:   Send the task argument
4: end for
```

Algorithm 15 xtasksCopyOut(kernelAddress)

```
1: for all Blocks in dataIn do
2:   if Block address == kernelAddress and Block state == Modified
     then
3:     Request the data block
4:     while Block state == Modified do
5:       Synchronize
6:     end while
7:   end if
8: end for
```

Chapter 7

Evaluation

In this chapter the different data reuse support designs have been analyzed with two real applications: Matrix Multiply and N-Body. Different versions of the applications have been evaluated in order to measure the potential benefits that can be achieved depending on the parallel strategy used in the applications. In some cases the parallel strategy is not the best one but helps to show the benefit that can be achieved depending on the application. Mostly all the experiments have been done using one accelerator, although for some of the cases two accelerators have been analyzed to observe how they can interfere in the data reuse.

7.1 Matrix Multiply

The baseline Tiled Matrix Multiply is shown in Listing 7.1. In particular, this sequential implementation helps to exploit data locality of one `AA` block (`Matmul_ai` label in the Figures).

On the other hand, in the OmpSss parallel version, function `matmulBlock` has been annotated as a task with target device `FPGA`, and therefore, every call to this function creates a task that goes to the pool of tasks, and later on taken by a thread in the thread pool, and issued to be accelerated in an accelerator within the `FPGA`. Directive `taskwait` in line 11 wait for all tasks created end. However, it is not guaranteed that the order of task creation is the same as the order of task issue and execution in the accelerators. Because of this, the exploitation of the data locality of block `AA` in the parallel version is not 100% guaranteed.

Another version evaluated, in order to force the exploitation of the data locality, is one with a pragma `taskwait` after the innermost-loop. This

```
1   for (i = 0; i < msize/BSIZE; i++) {
2      for (k = 0; k < msize/BSIZE; k++) {
3         ai = k*b2size + i*BSIZE*msize;
4         for (j = 0; j < msize/BSIZE; j++) {
5            ci = j*b2size + i*BSIZE*msize;
6            bi = j*b2size + k*BSIZE*msize;
7            matmulBlock(&AA[ai], &BB[bi], &CC[ci]);
8         }
9      }
10  }
11  #pragma omp taskwait
```

Listing 7.1: Tiled Matrix Multiply algorithm. Function `matmulBlock` performs matrix multiply of a tile of *BSIZE* × *BSIZE* elements.

version has been called `Matmul_ai_taskWait` in the Figures.

Finally, in order to analyze the benefit of avoiding output copies in the case of input output parameters, it has been also analyzed the Matrix Multiply version shown in Listing 7.2 (label `MatMul_ci` in the Figures). Like `Matmul_ai`, data locality exploitation is not guaranteed in the parallel version since the order of execution may differ from the order of issue. Then, another version has also been analyzed with a pragma `taskwait` after the innermost-loop, which has been called `Matmul_ci_taskWait` in the Figures.

```
1   for (i = 0; i < msize/BSIZE; i++) {
2      for (j = 0; j < msize/BSIZE; j++) {
3         ci = j*b2size + i*BSIZE*msize;
4         for (k = 0; k < msize/BSIZE; k++) {
5            ai = k*b2size + i*BSIZE*msize;
6            bi = j*b2size + k*BSIZE*msize;
7            matmulBlock(&AA[ai], &BB[bi], &CC[ci]);
8         }
9      }
10  }
11  #pragma omp taskwait
```

Listing 7.2: Tiled Matrix Multiply algorithm. Function `matmulBlock` performs matrix multiply of a tile of *BSIZE* × *BSIZE* elements, exploiting CC block locality

7.1.1 Results

Figures 27, 28, 31 and 32 show the overall execution time of the four different matrix multiply versions commented above. Each figure shows the execution time for different matrix sizes (x-axis) when using the original version of Nanos++ runtime, or one of the four different designs explored in this master thesis, all for one accelerator.

Figure 27 shows two important aspects: one is that all explored designs show improvements compared to the original code, being the relaxed version the one with more performance improvement. Second, although the order of the accelerated tasks is not guaranteed, the performance gain is, in average, of about 9-13% for all the sizes and designs.

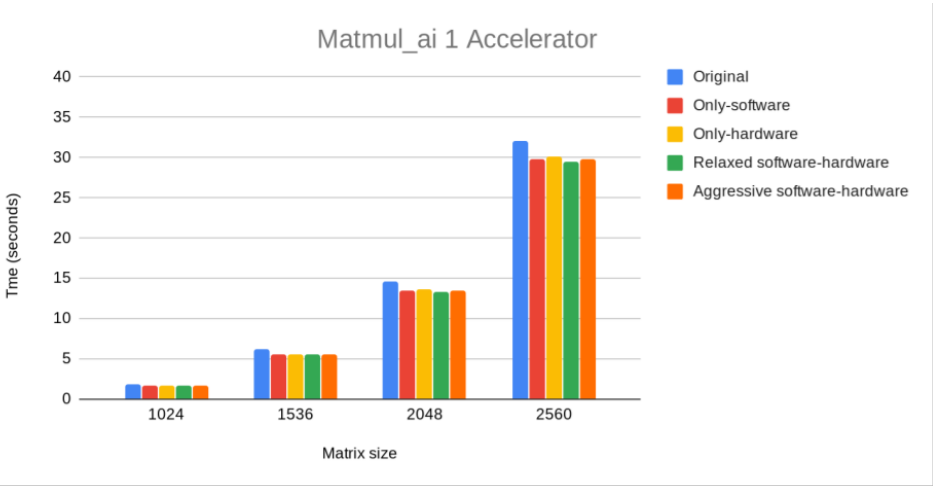

Figure 27: Elapsed time of OmpSs@FPGA version of Matrix Multiply with `AA` block reuse in the inner-most loop but with execution order of tasks not guaranteed.

The effect of forcing the task execution order to exploit `AA` block has been evaluated too. Figure 28 shows the execution time of the four different designs. The overall execution time has been increased in some cases and, in others, it is very similar. This is due to a co-lateral effect of the taskwait operation. When using taskwait, we are forcing invalidations, extra overhead, and forcing copies from kernel memory space to user memory space. In that case, the benefit of forcing reuse of `AA` block is hidden by the extra-overhead of keeping the information of output and copying then out. In particular, this is more critical in the case of the Aggressive Software-Hardware strategy

due to extra-overhead where different output `CC` blocks have to be flushed and updated in the directory inside the hardware.

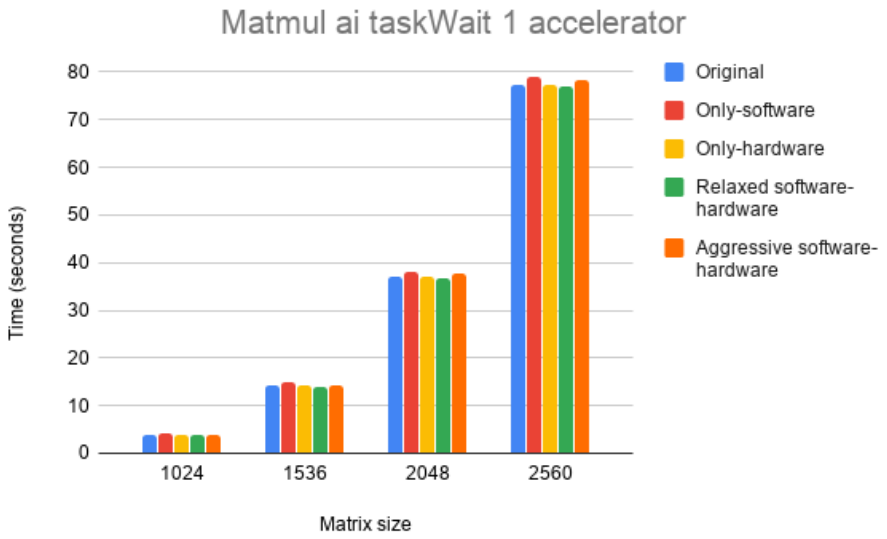

Figure 28: Elapsed time of OmpSs@FPGA version of Matrix Multiply with `AA` block reuse in the inner-most loop and forcing execution of inner-most loop tasks to be first.

Besides, Figure 29 shows a detailed execution trace of this configuration for a 1536×1536 matrix size, where it can be observed that the amount of input copies is reduced between the first task accelerated and the rest of tasks. The first task accelerated within the accelerator (last row of the timing view of Paraver) is one third bigger than the the rest of tasks accelerated. This is due to the `AA` block is not copied from the accelerator since it already has it. In fact, looking at the detail of the execution time dedicated to input copies, the original input copy execution time has been reduced from 4022407 to 2741111 microseconds ($1.5\times$ reduction of input copies) in the Relaxed Software-Hardware strategy.

In fact, looking at the detail of the execution time dedicated to input copies we have reduced the original input copy execution time from 4022407 to 2741111 microseconds ($1.5\times$ reduction of input copies) in the Relaxed Software-Hardware strategy. Figure 30 shows the overall execution time dedicated to input data copies to internal BRAM of the accelerators. All designs but Only-Hardware offer a significant reduction of the time dedicated

Figure 29: Execution trace of the Matrix Multiply for the `Matmul_ai_taskwait` case.

to input copies in the original version. The case of the Only-Hardware has to be analyzed in detail but it may be due to the lack of two ready tasks at the same time the Ready Task Queue.

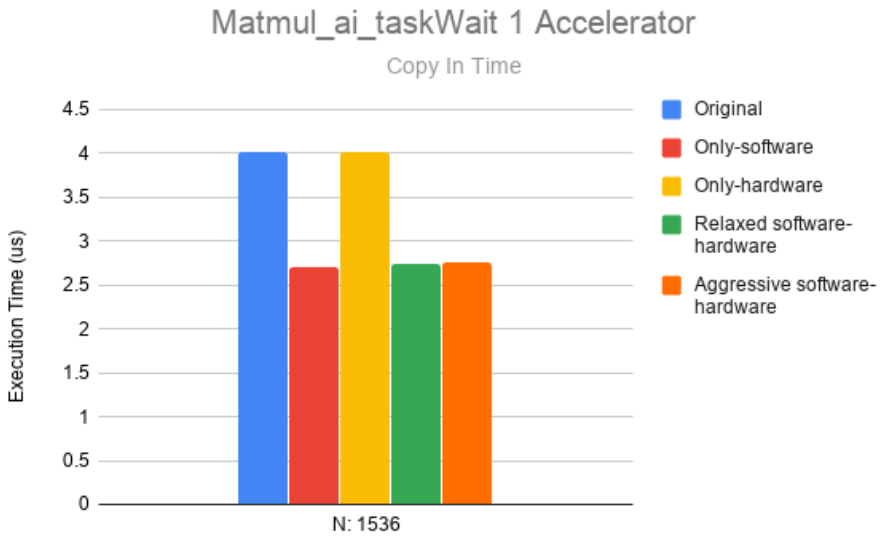

Figure 30: Elapsed time of input copy data of block `AA` for all the tasks accelerated in the FPGA.

Figure 31 shows that results of original and the proposed designs are similar. The reason is the same as the explained above for `MatMul_ai`: there is no way to ensure the order of the tasks execution, and then, the reuse of `CC` block is not guaranteed. However, if the task execution is forced to reuse the same block `CC` (see Figure 32) the performance improvement is significant. Unlike the case of Figure 28, here it is being exploited the reuse of input and also output of the SAME `CC` block in the same accelerator, meanwhile above the outputs were not reused and then it increased the invalidation and update overhead with the taskwaits. The average performance gain is of up to 14% in the cases with large matrix sizes. Indeed, as the `CC` block is an input/output parameter, the amount of input copies is reduced as it happens above. Figure 33 shows the execution time of the input data copies where it can be observed that the reduction is significant for all the designs but Only-Hardware.

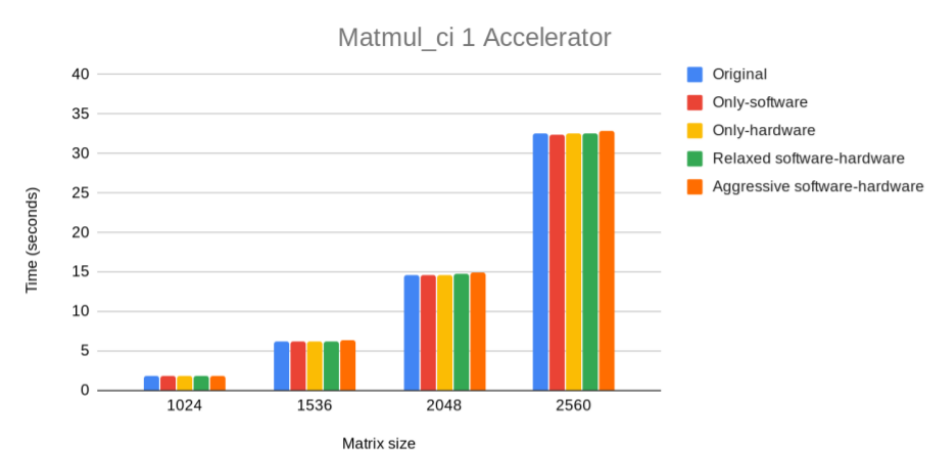

Figure 31: Elapsed time of OmpSs@FPGA version of Matrix Multiply with `CC` block reuse in the inner-most loop but with execution order of tasks not guaranteed.

The implemented designs have been tested for more than one accelerator in order to explore the new features of the Aggressive Software-Hardware functionalities. In those cases the performance results for both original and the different designs are very similar. Having two accelerators, with not data affinity scheduling policy at OmpSs runtime at accelerator level, tasks are assigned based on availability, breaking the possibility of reusing data already in an accelerator. Future work will be to include data affinity in

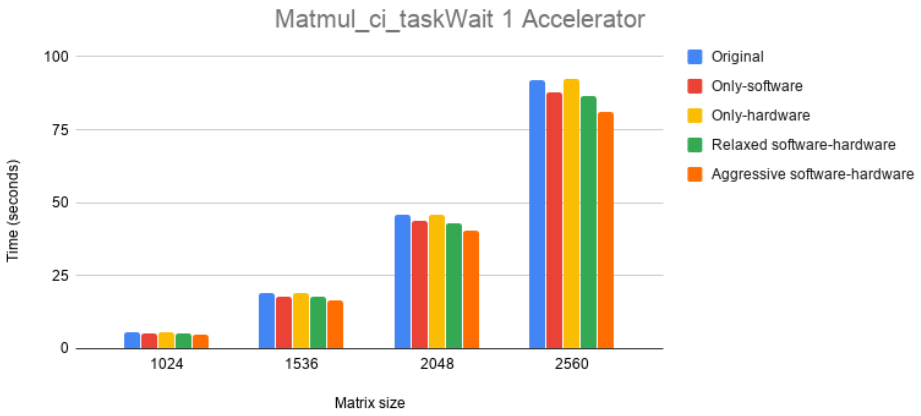

Figure 32: Elapsed time of OmpSs@FPGA version of Matrix Multiply with `CC` block reuse in the inner-most loop and forcing execution of inner-most loop tasks to be first.

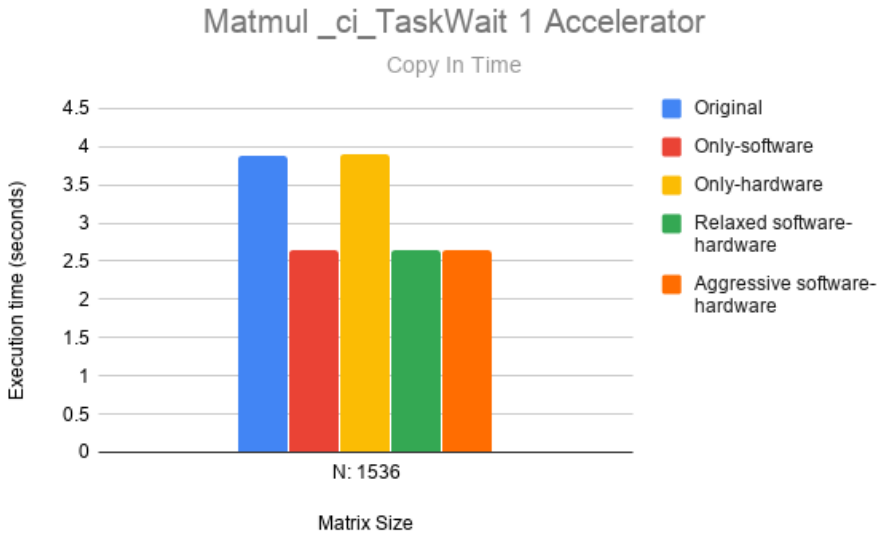

Figure 33: Elapsed time of input copy data of block `CC` for all the tasks accelerated in the FPGA.

the scheduling policy or perform a hardware re-mapping of the scheduling so that we map the task execution into the accelerator with the data.

7.2 N-body

The baseline n-body algorithm is shown in Listing 7.3. Function `solve_body` call to `calculate_forces` to update forces vector, and then, update the particules positions. In particular, we accelerate `calculate_forces` function making `calculate_forces_BLOCK` to be a task with target device `FPGA`. The original version of function `calculate_forces` helps to exploit data locality of one `b1` and `f0` memory blocks over the rest of particles blocks.

```
1  void calculate_forces (force_block_t * forces, particles_block_t
       * block1, particles_block_t * block2, const int size)
2  {
3     int i,j;
4     particle_fpga_ptr_t b1;
5     particle_fpga_ptr_t b2;
6     unsigned char * safe;
7     for (i = 0; i < size; i++) {
8        for (j = 0; j < size; j++) {
9           force_fpga_ptr_t f0 = (force_fpga_ptr_t)(forces+i);
10          b1 = (particle_fpga_ptr_t)(block1+i);
11          b2 = (particle_fpga_ptr_t)(block1+j);
12          safe = (unsigned char *)(b1 == b2 ? 1 : 0);
13          calculate_forces_BLOCK(f0, b1, b2, safe);
14       }
15    }
16   #pragma omp taskwait
17 }
18
19 void solve_nbody ( particles_block_t * __restrict__ particles,
20                    particles_block_t * __restrict__ tmp,
21                    force_block_t * __restrict__ forces,
22                    const int n_blocks,
23                    const int timesteps,
24                    const float time_interval ) {
25    int t;
26    for(t = 0; t < timesteps; t++) {
27       calculate_forces(forces, particles, particles, n_blocks);
28       update_particles(n_blocks, particles, forces,
      time_interval);
29    }
30 }
```

Listing 7.3: Tiled N-Body algorithm. Function `calculate_forces_BLOCK` performs the computation of the forces of one block of particules agaisnt another block of particules, and update the forces vector.

7.2.1 Results

Figure 34 shows the overall execution time for the N-Body algorithm for different input sizes (number of particles in x-axis) when using the original version of Nanos++ runtime, or one of the four different designs explored in this master thesis, all for one accelerator.

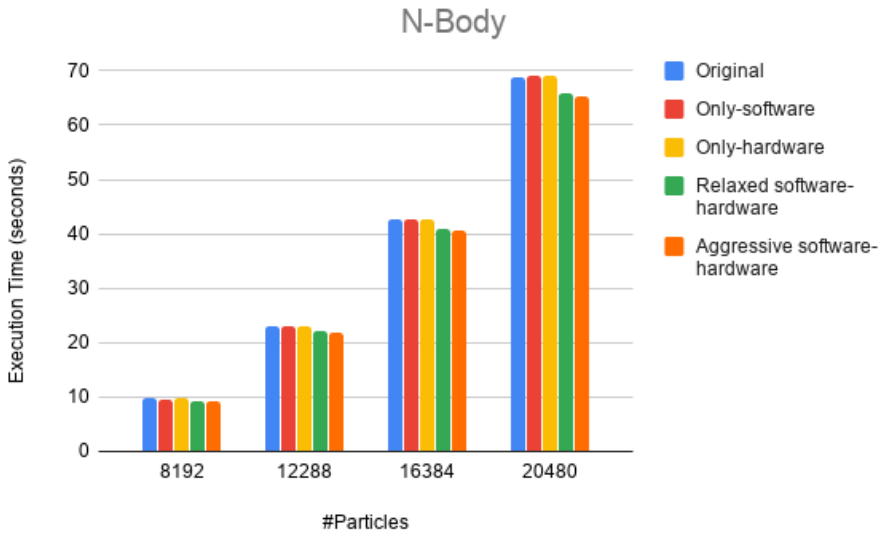

Figure 34: Execution Time of N-Body for the Original and proposed designs.

Like in the Matrix Mutltiply using taskwait to force the order of tasks, Figure 34 also shows performance gain for the Relaxed Software-Hardware and the Aggressive Software-Hardware versions. In those cases, it can take care of the copy out and invalidation, that do not affect the accelerator it already has the data, achieving more than 5% performance gain.

In the case of not using taskwait after the call, there is not way to guarantee the consecutive execution of tasks reusing the same data entry in the same accelerator. In that case, the execution time for the original and new proposals are very similar.

Chapter 8

Conclusions and future work

Finally, this chapter will explain the different conclusions that have been gathered from this master thesis. At the end, some future work that might be useful for extending the work done is presented too.

Once the results have been analyzed for each implemented design, and during all the process for the realization of this master thesis, there are several conclusions that have been extracted. Some of them are not directly connected to any specific design, but others depend on the design used and its implementation.

For example, it has been seen that in problems like the one proposed in this project the task affinity is very important. It can vary the performance of the application substantially, like seen in the matrix multiplication versions where it was forced the data reuse. Because of this, the solutions proposed may not fit in all the applications, but may mean big improvements on those where the same data is reused several times, such as Convolutional Neural Networks.

Moreover, an improvement on the logic block diagram has been explored with the utilization of different type of memories. The idea of having a closer memory to be used as a cache between the accelerators would be very beneficial for the purpose of this thesis, but it does not seem to be possible since there may be different array partitioning on the data of the accelerators. However, it has been demonstrated that BRAMs widths can be augmented, which enhances the memory transfer time.

More in detail, each design explored has its benefits and drawbacks. Regarding the ones designed independently, the only-software allows a time saving since the hardware does not need to be reconfigured. However, even though it is usually better than the original design, the only-software one

requires locks in several parts of Nanos++ that reduce the overall performance. Furthermore, it is even worse when the FPGA acceleration is faster than the task submission rate. Regarding the only-hardware, it depends too much on the tasks that are ready in the readyQueue BRAM, and it will never be possible to avoid output copies.

On the other hand, the hybrid designs are more flexible than the other two, but at the same time they imply more overhead on the TaskManager. The Aggressive software-hardware has been proved to be very beneficial in those cases where input output arguments are executed with good affinity. However, in those applications where this reuse is not so notorious, this design provokes an overhead that is even worse when there are different accelerators sharing data. For these cases, the Relaxed software-hardware design seems to be more suitable, since it does not implies an excessive overhead but is more flexible than the only-software and only-hardware ones.

To finalize, some enhancements that are beyond from this master thesis has been thought to try to improve the designs presented. Regarding the software part, as mentioned before, a better task scheduling could increase the task affinity and imply an improvement on the different designs, specially on the input output copies. Even more, it could be also designed a method to try to reorder the ready tasks in the readyQueue BRAM. Besides, the way the dataIn is accessed from Nanos++ in the Aggressive software-hardware design could be enhanced with a Pearson hash to reduce the number of accesses. Regarding the hardware part, the Aggressive software-hardware design could be also improved by copying out faster the requested data, and for example executing at the same time an FPGA task and a copy out task.

Bibliography

[1] Eriko Nurvitadhi et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 5–14. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021740. URL: http://doi.acm.org/10.1145/3020078.3021740.

[2] Jason Cong et al. "Understanding Performance Differences of FPGAs and GPUs". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2018, pp. 93–96. DOI: 10.1109/FCCM.2018.00023.

[3] Reiner Hartenstein. *The International Conference on Field-Programmable Logic, Reconfigurable Computing, and Applications*. URL: http://www.fpl.org/h/ (visited on 09/18/2019).

[4] Murad Qasaimeh et al. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels". In: *The 15th IEEE International Conference on Embedded Software and Systems*. May 2019. DOI: 10.1109/ICESS.2019.8782524.

[5] Umer Farooq, Zied Marrakchi, and Habib Mehrez. "FPGA Architectures: An Overview". In: *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. Springer New York, 2012, pp. 7–48. ISBN: 978-1-4614-3594-5. DOI: 10.1007/978-1-4614-3594-5_2. URL: https://doi.org/10.1007/978-1-4614-3594-5_2.

[6] Yecheng Lyu, Lin Bai, and Xinming Huang. "Real-Time Road Segmentation Using LiDAR Data Processing on an FPGA". In: Nov. 2017. DOI: 10.1109/ISCAS.2018.8351244.

[7] Federico Angiolini et al. “1024-Channel 3D ultrasound digital beamformer in a single 5W FPGA”. In: Mar. 2017, pp. 1225–1228. DOI: 10.23919/DATE.2017.7927175.

[8] Weiqiang Liu et al. “Optimized Schoolbook Polynomial Multiplication for Compact Lattice-Based Cryptography on FPGA”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* PP (June 2019), pp. 1–5. DOI: 10.1109/TVLSI.2019.2922999.

[9] Guido Stanchieri et al. “An FPGA-Based Architecture of True Random Number Generator for Network Security Applications”. In: Jan. 2018. DOI: 10.1109/ISCAS.2018.8351376.

[10] Ran Shu et al. “Direct Universal Access: Making Data Center Resources Available to FPGA”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 127–140. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/shu.

[11] Chethan Ramesh et al. “FPGA Side Channel Attacks without Physical Access”. In: Apr. 2018, pp. 45–52. DOI: 10.1109/FCCM.2018.00016.

[12] Barcelona Supercomputing Center. *BSC-CNS — Barcelona Supercomputing Center - Centro Nacional de Supercomputación*. URL: https://www.bsc.es/ (visited on 10/15/2019).

[13] Barcelona Supercomputing Center. *The OmpSs Programming Model*. URL: https://pm.bsc.es/ompss (visited on 09/18/2019).

[14] *European Project for the Holistic Development of Exa-Scale Supercomputer and Data Centre Technologies*. URL: https://euroexa.eu/ (visited on 09/18/2019).

[15] *OmpSs@FPGA*. URL: https://pm.bsc.es/ompss-at-fpga (visited on 09/18/2019).

[16] Yufei Ma et al. “Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks”. In: Feb. 2017, pp. 45–54. DOI: 10.1145/3020078.3021736.

[17] Zhiyuan Shao et al. “Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching”. In: Feb. 2019, pp. 320–329. DOI: 10.1145/3289602.3293900.

[18] L. Sommer, J. Korinth, and A. Koch. "OpenMP device offloading to FPGA accelerators". In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. July 2017, pp. 201–205. DOI: 10.1109/ASAP.2017.7995280.

[19] Roberto Rigamonti et al. "Transparent Live Code Offloading on FPGA". In: *CoRR* abs/1609.00130 (2016). arXiv: 1609.00130. URL: http://arxiv.org/abs/1609.00130.

[20] Tomasz Patyk et al. "Design Methodology for Offloading Software Executions to FPGA". In: *Signal Processing Systems* 65 (Nov. 2011), pp. 245–259. DOI: 10.1007/s11265-011-0606-x.

[21] Ángel Álvarez et al. "OpenMP Dynamic Device Offloading in Heterogeneous Platforms". In: Aug. 2019, pp. 109–122. ISBN: 978-3-030-28595-1. DOI: 10.1007/978-3-030-28596-8_8.

[22] M. Knaust, F. Mayer, and T. Steinke. "OpenMP to FPGA Offloading Prototype Using OpenCL SDK". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2019, pp. 387–390. DOI: 10.1109/IPDPSW.2019.00072.

[23] Jialiang Zhang and Jing Li. "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 25–34. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021698. URL: http://doi.acm.org/10.1145/3020078.3021698.

[24] Yuxin Wang et al. "An integrated and automated memory optimization flow for FPGA behavioral synthesis". In: *17th Asia and South Pacific Design Automation Conference*. Jan. 2012, pp. 257–262. DOI: 10.1109/ASPDAC.2012.6164955.

[25] Pedro Becker et al. "A Low-Cost BRAM-Based Function Reuse for Configurable Soft-Core Processors in FPGAs". In: Jan. 2018, pp. 499–510. ISBN: 978-3-319-78889-0. DOI: 10.1007/978-3-319-78890-6_40.

[26] Xilinx. *Xilinx Zynq-7000 SoC ZC706 Evaluation Kit*. URL: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html (visited on 09/29/2019).

[27] Barcelona Supercomputing Center. *Mercurium*. URL: https://pm.bsc.es/mcxx (visited on 09/29/2019).

[28] Barcelona Supercomputing Center. *Nanos++*. URL: https://pm.bsc.es/nanox (visited on 09/29/2019).

[29] Xilinx. *Vivado Design Suite - HLx Editions*. URL: https://www.xilinx.com/products/design-tools/vivado.html (visited on 09/29/2019).

[30] Barcelona Supercomputing Center. *Extrae*. URL: https://tools.bsc.es/extrae (visited on 09/29/2019).

[31] Barcelona Supercomputing Center. *Paraver*. URL: https://tools.bsc.es/paraver (visited on 09/29/2019).

[32] Xilinx. *Integrated Logic Analyzer (ILA)*. URL: https://www.xilinx.com/products/intellectual-property/ila.html (visited on 09/29/2019).

[33] Peter K. Pearson. "Fast Hashing of Variable-length Text Strings". In: *Commun. ACM* 33.6 (June 1990), pp. 677–680. ISSN: 0001-0782. DOI: 10.1145/78973.78978. URL: http://doi.acm.org/10.1145/78973.78978.